Introducere in programare

C

CAPITOLUL 1 Introducere in programare

1. Introducere(

Calculatoarele electronice au apărut în jurul anului 1940. Totuşi unele maşini mecanice de calculat au fost propuse anterior:

- Pascal 1641 – realiza numai adunare şi scădere
- Leibnitz 1674 – realiza şi înmulțire
- Babbage 1840 – maşină analitică

Chiar şi în aceste aparate primitive unele din componentele unui arhitecturi moderne pot fi găsite. De exemplu un registru temporar de memorie care menține variabilele aflate în curs de procesare. Acest registru este cunoscut sub denumirea de registru acumulator. Calculatoarele electronice au fost inițial propuse sub forma unui aparat abstract care poate realiza sau simula orice tip de maşină mecanică. Forma teoretică a fost introdusă de Alan Turing pe la mijlocul anului 1930.

Ca şi în cazul multor alte tehnologii, dezvoltarea practică a unei maşini care să respecte acest model, a fost inflențată de al doilea război mondial. Acest lucru a fost posibil deoarece se dorea calcularea traiectoriilor de rachete (Collosus şi Eniac) sau decodarea codului Enigma folosit de Germania pentru protecția mesajelor.

Primul program stocat electronic a fost folosit în 1948. În primii ani de dezvoltare (1950) computerele şi impactul lor ulterior în viața umană au fost mult subevaluate. E de notorietate declarația lui Bill Gates, fondatorul Microsoft cum că nu vede cum ar putea fi vreo dată nevoie de mai mult de 640 Ko RAM.

În anii ’70 a apărut pastila de siliciu şi tehnologia VLSI (Very large Scale Integration) ca tehnică de producție fapt ce adus la computere ieftine şi rapide. Acest lucru a condus la creşterea şi diversificarea rapidă a categoriilor de utilizatori care ăşi puteau permite să le achiziționeze.

Perioada 1970-1980 care este caracterizată de prin apariția microsistemelor pe 8 biți, care deşi erau scumpe permiteau accesul separat de *main-frame* la posibilitățile sistemului de calcul. În această perioadă este demnă de remarcat apariția în 1973 a lui I8080 care poate fi considerată un punct de referință în domeniu. Din acest moment softul a început să fie dezvoltat nu numai pentru calcule extrem de complexe ci şi pentru jocuri, editoare de texte, programe de baze de date şi alte facilități cerute de piață.

Tot în această perioadă, forțată şi de ruperea de main-frame, a pornit şi dezvoltarea sistemelor de operare "locale" CPM şi apoi DOS.

Sistemul de operare MS-DOS a fost realizat prin adaptarea la noile instrucțiuni maşină a unui *microkernel* de UNIX la care s-au adăugat funcțiile necesare controlului perifericelor care începuseră să fie ataşate calculatoarelor personale.

Din 1980 până în 1985, deşi încă foarte scumpe, au început să apară pe piață arhitecturi din ce în ce mai puternice pe 16, 32 biți iar aria lor de folosire s-a extins foarte mult şi în alte domenii.

Nu este de neglijat nici impactul produs de interfețele vizuale şi de controalele prin zone active ale ecranului. Aceste controale care sunt activate din *mouse* sunt caracteristice sistemului de operare WINDOWS.

Tot în această perioadă, din raţiuni economice şi practice, s-a produs dispariţia microcalculatoarelor acestea fiind mult depăşite de cerinţele pieţii.

Din 1985 se poate spune că s-a intrat într-o nouă eră în acest domeniu prin ieftinirea dramatică a hardware-ului şi apariţia reţelelor de calculatoare.

O dată cu ieftinirea arhitecturilor de calcul s-a putut divesrifica şi mai mult aria de utilizare a lor. Acest lucru a avut ca rezultat o creştere a preţului softului.

Conceptul de reţea permitea transferuri rapide de date , a căror cantitate a început să fie din ce în ce mai mare.

Iniţial majoritatea reţelelor de calculatoare erau "închise " în sensul că nu comunicau între ele. Apoi în ultima perioadă termenul de sistem "deschis" capătă din punct de vedere practic o răspândire din ce în ce mai mare la nivel global. Din acest punct de vedere reţele de tip Arpanet , Internet ş.a.m.d sunt reprezentative.

În ultimii ani însuşi conceptul de transmisie de informaţie interumană tinde să se schimbe. Televiziunea digitală este deja implementată , iar preţul perifericelor dedicate a scăzut extraordinar astfel încât se preconizează dezvoltarea calculatorului astfel încât el să nu mai fie strict orientat spre aplicaţii dedicate ci să fie o legătură dinamică de orice tip (audio, video, teleworking) a individului cu societatea.

Vom prezenta câteva din aplicaţiile mai cunoscute ale calculatoarelor în diverse domenii:

- Industrie: control şi automatizare → microcontrolere, CAM (Computer Aided Manufactory);
- Economie: gestiune, transferul banilor, "banii electronici" .
- Medicină: diagnoză automată, cercetarea electronică la diverse nivele ale organismului uman, analiza codului genetic, diagnoză automată → reţele neurale, staţii grafice, sisteme expert.
- Pictură: noi ramuri ale picturii folosind computerul, restaurare → holografie reţele neurale, procesoare analogice.
- Muzică şi televiziune: prelucrări de imagini şi sunete practic de orice natură → procesoare analogice, staţii grafice, arhitecturi MMX .
- Proiectare : programele de tip CAD (Computer Aided Design)

Se mai pot da exemple în multe domenii, însă este cert că la ora actuală viaţa şi societatea umană nu mai pot fi concepute fără calculator.

1.1. Programarea şi limbaje de programare(

Prin **programare** se înțelege în mod generic transpunerea unor operații repetitive, asupra unui set de date, într-un limbaj inteligibil de către un sistem de calcul care urmează ulterior să le execute. Acest lucru este realizat în două etape

- etapă în care este implicat omul şi anume cea de trecere de la problema reală la transpunerea într-un limbaj de programare.
- o a doua etapă, automată, care transpune **codul (sursă (** înşiruirea de instrucțiuni specifice limbajului respectiv**)** într-un **cod direct executabilă (**inteligibil sistemului de calcul) lucru de care se ocupă programe specializate numite **compilatoare**.

Rolul (programării este ca fie dată o anumită operațiune sau suită de operațiuni repetitivă(e) care se aplică asupra unor seturi de date mereu diferite să fie scris un program care să ceară setul de date **de (intrare**(cele care trebuie să fie prelucrate) să execute asupra lor suita standard de operațiuni şi să livreze **datele (de (ieşire (** adică rezultatele).

În aceste condiții programul trebuie să fie compus din mai multe instrucțiuni. În primul rând cele care **rezervă** (declară o zonă de memorie ca fiind special dedicată numai pentru anumite date), o zonă de memorie în care se vor citi datele de intrare, o zonă de memorie în care se vor păstra datele **intermediare** (cele care apar în decursul calculelor dar de care utilizatorul nu are nevoie) şi de asemeni pe cele de ieşire. Programul trebuie să conțină instrucțiuni care să citească de la utilizator datele de intrare, urmate în sfârşit de instrucțiunile de calcul propriu zis ca în final să conțină instrucțiunile de afişare a rezultatelor (datele de ieşire)

1.2(Arhitectura unui sistem de calcul(

Ca ***observație*** de bază: un sistem de calcul este pur şi simplu versiunea mult dezvoltată a unui calculator de buzunar şi deci nu trebuie să ne aşteptăm de la el ca vreo dată să facă altceva decât l-a pus cineva, un programator prin intermediul unui program, să facă. “Inteligența” aparentă a unui sistem de calcul provine din cantitatea de inteligență umană investită în respectivul program aflat în execuție.

Hilar, am putea spune că omul încearcă să “facă pe cineva după chipul şi asemănarea lui” dar rezultatul este o copie palidă care altfel este extrem de utilă. Din aceste lucruri derivă vestitul principiu de aur al primilor programatori GIGO (Garbage In Garbage Out), care într-o traducere prozaică înseamnă: “gunoi bagi gunoi scoți” şi care are meritul, în afară de cel umoristic, de a atrage atenția oricui care se va apuca de programare că din nefericire calculatorul face numai ce i se indică prin diverse metode să facă. De aici rezultă o primă atenționare valabilă aprope tot timpul în cazul **depanării**(execuției pas cu pas în scopul găsirii eventualelor defecte în funcționare) unui program : CAUTĂ ÎNTÂI EROAREA TA DE PROGRAMATOR ŞI APOI DĂ VINA PE CALCULATOR.

Desigur există un procent cam de 5% din situații în care un program scris corect este executat greşit, lucru care apare ca rezultat al interferenței unor programe neautorizate, cum ar fi viruşii, cu programul aflat în execuție. În cazul în care nu este

valabil acest lucru problema este mai complexă putând fi implicate defecţiuni aleatorii ale părţii hardware. Oricum ar fi aceste lucruri intră în sfera de acţiune a unui inginer de sistem, şi deci nu sunt luate aprioric în consideraţie fiind de obicei rare.

Pentru a putea înţelege arhitectura vom căuta anumite analogii între un om şi un sistem de calcul.

Să analizăm ce se întâmplă cu un om în momentul în care vede o formulare de genul : 2*2=… .

Deşi ni se pare ceva evident, cu toţii am învăţat pe de rost tabla înmulţirii să încercăm acum o descompunere a paşilor care sunt executaţi.

În primul rând memorăm în memoria temporară (sau de scurtă durată) **operanzii(**datele asupra cărora vom executa o operaţiune**)** apoi realizăm operaţiunea de înmulţire rezultând 4, care este până la comunicarea lui verbală sau prin scris tot în memoria temporară. Se observă că operanzii 2,2 şi operaţiunea * au fost preluate de pe un suport extern si anume cel scris, rezultatul putând fi **returnat(**trimis către emiţătorul unei cereri de lucru **),** după cum am spus tot pe acelşi sistem extern de stocare.

Acest exemplu nu a fost dat fără un scop şi anume, rolul lui este printre altele de a vă obişnui cu nivelul extrem de scăzut la care se face analiza şi descompunerea unei probleme reale în momentul în care doresc să o transpun într-un limbaj de programare.

Şi acum să analizăm **arhitectura bloc** (structura funcţională) a unui sistem de calcul.

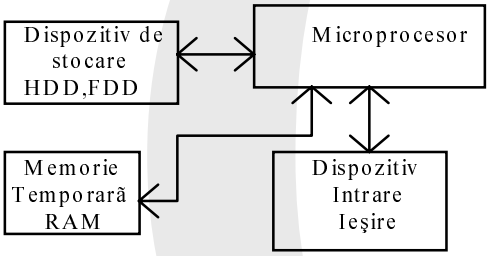


Fig. 1 Arhitectura unui sistem de calcul

După cum se observă din fig. 1 într-un sistem de calcul există:

- un dispozitiv de stocare în masă, rolul caietului pe care scrim /citim în cazul nostru. Acest dispozitiv este materializat prin Hard Disk (HDD) căruia i se mai spune şi suport fix de date şi Floppy Disk (FDD) căruia i se mai spune şi suport mobil de date .
- o memorie temorară, RAM (Random Acces Memory) care joacă exact rolul memoriei de scurtă durată umane.
- dispozitive de Intrare sau de Ieşire prin care sistemul primeşte sau returnează date. În cazul unui sistem de calcul acestea sunt tastatura şi mouse ca dispozitive de intrare şi monitorul sau imprimanta ca dispozitive de ieşire.

A mai rămas de lămurit problema microprocesorului care este “inima” unui sistem de calcul şi are rolul de a coordona întregul sistem. Desigur că această coordonare se realizează tot prin intermediul unor programe care există în ceea cea am putea numi, prin analogie cu omul, partea de reflexe condiţionate, aceste programe se găsesc într-o memorie ROM (Read Only Memory) deci nu pot fi decât citite.

1.3(Funcţionarea unui sistem de calcul(

Sistemul de calcul după cum se observă comportă destul similitudini cu omul la nivel de arhitectură şi deasemeni, după cum vom discuta, şi la nivel de funcţionare.

Să vedem cum se realizează, bineînţeles la nivel simplificat, execuţia unei formule de genul 2*2=… .

Se citeşte de pe suportul de stocare programul care conține cererea noastră, se încarcă în memoria temporară (de acest lucru se ocupă **sistemul de operare(**programul specializat care se ocupă de traducerea unor instrucțiuni simple şi intuitive ale utilizatorului în comenzi directe către partea de execuție a sistemului**)**, în urma cererii noastre explicite de lansare a programului) şi se începe execuția lui de către procesor execuție care urmează aproximativ sistemul uman de gândire. Pe scurt modul de execuție la nivelul procesorului poate fi descris ca :

- citeşte datele utile (din RAM în aceast caz) (2,2)
- citeşte operația care trebuie executată asupra datelor suspomenite(tot din RAM)(*)
- execută operația (2*2=4)
- returnează rezultatul (înnapoi în RAM)(4).

La nivelul sistemului de calcul modul de execuție arată ca mai jos:

- citeşte programul de executat (de pe sistemul de stocare fix/mobil) şi
- încarcă-l în RAM
- execută programul (din RAM)
- la terminarea programului aşteaptă noua cerere de execuție

Memoria Temporară a unui sistem de calcul(

După cum am pomenit în primul capitol, una din componentele de bază ale unui sistem de calcul este memoria dinamică RAM sau memoria temporară.

Această memorie a fost concepută ținând cont de faptul că sistemul de numerație folosit în interiorul unui calculator este bază 2. De aici rezultă că orice fel de dată sau instrucțiune efectiv procesată sau executată la nivelul maşinii trebuie să fie pastrată sub forma ei binară (deci corespunzătoare bazei 2). Întrucât în baza 2 nu există decât două numere 0 şi 1 rezultă că vom avea reprezentări sub forma unor şiruri de 0 şi de 1. La nivelul maşinii materializarea efectivă a respectivelor numere este realizată, de obicei, prin ataşarea logică a lui “1” la o tensiune de 5 volți iar lui “0” lipsei acestei tensiuni. Totuşi indiferent de nivelul tehnologic este absolut nepractic ca să considerăm fix 5 V sau fix 0 V ca valori de comparație pentru că acest lucru ar fi dus la nişte echipamente extrem de scumpe şi de lente. Din aceasta cauză se consideră a fi 0 sau 1 logic în jurul valorilor sus menționate cu $\pm\Delta V$. De aici a rezultat posibilitatea folosirii unui condensator ca element de memorare. Acest lucru este posibil deoarece în cazul unei reîncărcări periodice a acestuia se poate obține ca tensiunea la bornele lui să varieze în limitele necesare nouă dacă fac suficient de repede această reîncărcare. În aceste condiții foarte simplist putem vedea o memorie dinamică ca o matrice imensă de condensatoare care pot fi încărcate sau descărcate în cursul operațiunilor de scriere/citire.

Aici mai apare o problemă şi anume numărul de biți care compun datele utile de calcul (**bit (**unitate informațională corespunzătoare existenței sau a unei tensiuni la intrarea unui dispozitiv digital) care pot fi procesați la un moment dat de un procesor.

Întrucât memoria RAM este cea care furnizează respectivele date era logic ca ea să fie organizată în diviziuni de o mărime egală cu cea a numărului de biți sus menționați. Aceste grupări se numesc **locații de memorie(**şi pot fi considerate ca minima unitate care poate fi scrisă sau citită dintr-o memorie. O dată lămurit acest aspect devine clar că pentru a mă putea referi la una sau alta din locații trebuie să existe o metodă de individualizare strictă. S-a ales cea mai simplă cu putință şi anume numerotarea lor numărul locației numindu-se **adresa locației**.

1.4. Noțiunea de algoritm(

În procesul de rezolvare a unei probleme folosind calculatorul există două faze:

- definirea şi analiza problemei;
- proiectarea şi implementarea unui algoritm care rezolvă problema.

Definirea şi analiza problemei poate fi la rândul ei descompusă în:
- specificarea datelor de intrare;
- specificarea datelor de ieşire.

Specificarea datelor de intrare constă în:

1. Ce date vor fi primite la intrare;
2. Care este formatul (forma lor de reprezentare) a datelor de intrare;
3. Care sunt valorile permise sau nepermise pentru datele de intrare;
4. Există unele restricții (altele decât la 3) privind valorile de intrare;
5. Câte valori vor fi la intrare, sau dacă nu se poate specifica un număr fix de valori, cum se va şti când s-au terminat de introdus datele de intrare.

Specificarea datelor de ieşire trebuie să țină cont de următoarele aspecte:

1. Care din valorile rezultate în cursul aplicării algoritmului de calcul, asupra datelor de intrare, vor fi afişate (necesare utilizatorului). În acest pas se face diferențierea clară între date intermediare şi date de ieşire.
2. Care va fi formatul datelor de ieşire (de exemplu un număr real poate fi afişat cu trei sau cu cinci zecimale, sau un text poate fi afişat integral sau parțial)
3. Sunt sau nu necesare explicații suplimentare pentru utilizator în afara datelor de ieşire.
4. Care este numărul de date de ieşire care trebuie transmise către ieşire.

O definiție a noțiunii de algoritm poate fi: înlănțuirea de paşi simpli, operații distincte care descriu modul de prelucrare a unor date de intrare în scopul rezolvării unei probleme. Deci, ALGORTIMUL ESTE ACEA PARTE DE CALCUL COMUNĂ ÎN CAZUL REZOLVĂRII UNEI PROBLEME DE CALCUL.

Un exemplu simplu de algoritm ar fi suita de operații matematice făcută în rezolvarea unei ecuații matematice de gradul II $aX^2+bX+c=0$, coeficienții a, b, c se schimbă dar modul de procesare a valorilor lor, nu.

Proprietățile unui algoritm sunt:

1. Este compus din instrucțiuni simple şi clare.
2. Operațiunile specificate de instrucțiuni se execută într-o anumită secvență.
3. Soluția trebuie obținută într-un număr finit de paşi.

Din cele spuse mai sus rezultă că UN ALGORITM ESTE INDEPENDENT DE TIPUL DE LIMBAJ ÎN CARE ESTE TRANSPUS SAU DE TIPUL DE MAŞINĂ PE CARE ESTE EXECUTAT.

Observaţie: În această carte se vor discuta numai ALGORITMI SECVENŢIALI şi respectiv PROGRAMAREA SECVENŢIALĂ.

Un algoritm secvenţial este acel algoritm creat pentru execuţia pe un calculator secvenţial. Denumirea de secvenţial provine din faptul că acesta nu poate executa decât o singură operaţie la un moment dat.

1.5. Reprezentarea algoritmilor(

În general, un algoritm poate fi considerat ca o descriere a prelucrării efectuate asupra unui flux de date, prelucrare care are loc cu un scop bine determinat. Ca de obicei, în cazul dezvoltării unei noi teorii, s-a căutat atât o cale de standardizare a modului de descriere a unui algoritm, cât şi o cale de a-l face independent de un limbaj de programare anume rezultând două metode clasice:

- metoda schemei logice (diagramei de flux (flowchart));
- metoda pseudocodului.

1.5.1. Metoda schemei logice(

În cadrul acestei metode se foloseşte un set de simboluri, prezentat în figura 1.1, pentru descrierea paşilor ce trebuie executaţi pentru ca programul rezultat să ne rezolve o anumită problemă.

Deşi a fost extrem de folosită, până nu de mult, această metodă pare a pierde teren în faţa reprezentării de tip pseudocod, poate şi datorită timpului suplimentar pierdut de utilizator cu executarea simbolurilor grafice.

Să analizăm un program de calcul a mediei pentru trei note şi să vedem cum ar apărea descris prin această metodă.

După cum se observă în figura 1.2 se descriu efectiv pas cu pas toate operaţiunile care trebuiesc urmate.

În figura 1.2 s-a specificat şi începutul programului, fapt care pare inutil, dar în cazul descrierii unui program foarte mare se prea poate ca schema logică să se întindă pe zeci de pagini, caz în care devine evidentă necesitatea declarării începutului şi sfârşitului de program.

În următorul pas se declară variabilele. O variabilă este un identificator propriu al programatorului, care în cazul operaţiunii de declarare se ataşează, la generarea programului, unei zone de memorie a cărei dimensiune este în directă concordanţă cu tipul de dată stocat de aceasta.

Se trece apoi la citirea datelor de intrare. În acest caz este vorba de trei note ceea ce presupune că valorile de intrare pot avea şi cu zecimale. Mai târziu se va discuta despre alegerea tipului de variabilă funcţie de specificaţiile utilizatorului.

Se execută calculul propriu zis, după care se afişează rezultatul cu 2 zecimale. A mai rămas doar marcarea sfârşitul de program. Chiar dacă metoda are o impresie "artistică" sporită câte o dată se pierde prea mult timp şi hârtie în cazul folosirii ei şi de aceea din ce în ce mai mulţi preferă ca metodă de descriere pseudocodul.

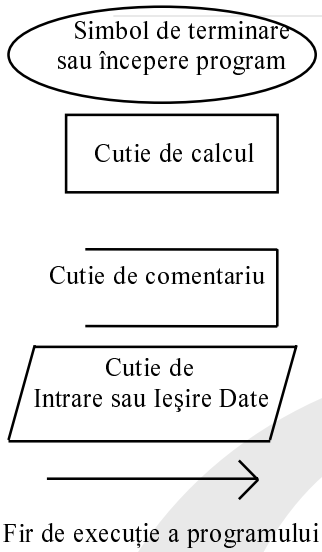


Fig. 1.1. Simboluri standard pentru metoda schemei logice

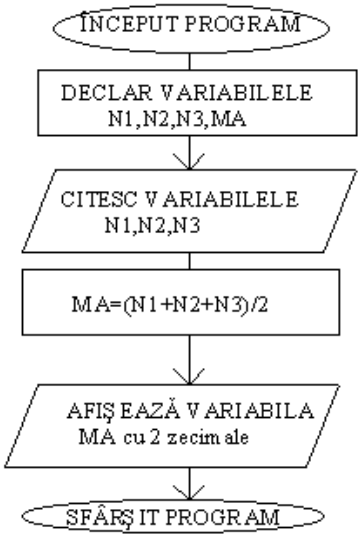


Fig. 1.2. Schema logică unui program de calcul a mediei pentru trei note

1.5.2. Metoda pseudocodului(

Față de schema logică are dezavantajul că nu întotdeauna redă absolut toți paşii de calcul. Totuşi este mai comodă de folosit în cazul în care se descriu algoritmi pentru limbajele de programare de nivel mediu şi înalt.

Schema logică este cea mai indicată ca metodă de descriere, în special, în cazul implementării unui algoritm folosind limbaj de asamblare, unde descrierea amănunțită a fiecărui pas este necesară pentru putea minimiza şansele apariției unor erori de implementare.

Descrierea efectivă a paşilor ce trebuie efectuați, folosind cuvintele limbii române împreună cu cuvinte cheie ale limbajului de programare în care se va implementa algoritmul se numeşte pseudocod. Se observă că în acest caz criteriul independenței totale a descrierii față de limbajul de implementare nu este întru totul respectat.

Pentru comparație vom descrie mai jos în pseudocod problema prezentată în figura 1.2.

```
variabile: ma,n1,n2,n3
{
citeşte n1,n2,n3
ma=(n1+n2+n3)/3
afişează ma cu 2 zecimale
}
```

2. Limbajul C

Limbajele de programare de nivel mediu au fost serios dezvoltate pe la mijlocul anilor ’50. Ideile de bază folosite pentru proiectarea lor derivă din formulele funcționale λ-Church ce fuseseră propuse prin anii ’30. La ora actuală se estimează că există peste 2000 de limbaje de programare, diferențele între ele fiind legate în principal de stilul de programare.

Limbajul FORTRAN (FORmula TRANslator) a fost unul dintre primele limbaje de programare. El este folosit şi acum în industrie sau cercetare pentru a implementa aplicații specifice cu volume mari de calcule matematice complexe.

Pe lângă limbaje de programare pentru maşini secvențiale, s-au dezvoltat şi limbaje care suportă procesare paralelă. Din nefericire majoritatea nu sunt portabile, fiind orientate strict pe arhitectura maşinii pentru care a fost dezvoltat limbajul. De abia în ultimii zece ani dezvoltarea rapidă a rețelelor de calculatoare a condus şi la o încercare de standardizare a unor limbaje ce pot crea cod executabil simultan pe mai multe sisteme de calcul.

Avantajele programării la nivel inalt sunt:

- ascund diferențele de arhitectură internă
- sunt uşor de urmărit şi menținut (modificare şi depanare uşoară)
- sunt portabile . De exemplu un cod scris în C poate fi executat şi sub UNIX şi sub DOS
- nu au cel mai eficient cod, în comparație cu cel scris in ASM direct

De obicei sistemele de operare erau dezvoltate in limbaj de ansamblare pentru a fi mai rapide . Limbajul C, dezvoltat in 1972 este Dennis M.Retchie la Laboratoarele AT&T Bell, este primul limbaj pentru crearea sisteme de operare. Numele limbajului provine din faptul că este rezultatul îmbunătățirii limbajului B, folosit în scrierea UNIX pentru DEC PDP7. Prima documentație despre acest limbaj a fost “The C Programing Language”, scrisă de Retchie şi Brian Kernighan în 1977. Înaintea ei exista doar “The C Reference Manual”, scrisă de Retchie. O caracteristică importantă a acestui limbaj este faptul că poate fi considerat simultan şi un limbaj de nivel mediu şi un limbaj de nivel scăzut.

O dată cu răspândirea lui printre programatorii profesionişti s-a trecut şi la adaptarea compilatorului pentru sistemul de operare MS-DOS. De asemenea a apărut şi altă problemă şi anume faptul că standardul K&R nu mai era satisfăcător. De asemenea datorită variantelor de compilare pentru PC-uri C începuse să-şi piardă din portabilitate.

În acel moment Institutul Național pentru Standarde al Americii (ANSI), a creat un subcomitet pt. a defini o versiune oficială pt. limbajul C, rezultând astfel ANSI C.

Limbajul C şi versiunile sale OOP (Object Oriented Programming) C++ şi mai noul Visual C++ sunt printre cele mai folosite limbaje de programare la ora actuală. A fost proiectat pornind de la două limbaje dezvoltate spre sfârşitul anilor ’60: BCPL şi prima sa versiune B.

Un limbaj de nivel scăzut este orientat pe arhitectură şi poate fi definit ca având facilitatea de a scrie instrucțiuni ce se referă direct la elemente ale respectivei arhitecturi interne (ca în cazul limbajului de asamblare).

Un limbaj de nivel înalt sau mediu este orientat pe problemă permițând programatorului să-şi structureze programul cât mai apropiat posibil de problema ce trebuie implementată. Acest lucru scade şansele unei erori de proiectare, la nivel logic, al aplicației. Chiar dacă programul va fi mai ineficient din punct de vedere al vitezei şi al ocupării resurselor (prin comparație cu un program scris în limbaj de asamblare) el are avantajul că poate fi scris repede şi extins cu uşurință.

Limbajul C permite folosirea ambelor tehnici: programare structurată şi acces direct la maşină, fapt care-l face să fie foarte flexibil.

Ultimul şi poate cel mai important motiv pentru învățarea limbajului C este faptul că permite trecerea cu uşurință, la C++ sau Java.

Deşi limbajul **C** are o libertate foarte mare in scrierea codului, el acceptând multe forme de scriere care în alte limbaje nu sunt premise, acest lucru ascunde însă şi pericole, în special pentru programatorii fără experiență.

2.1 Structura generică a unui program scris în C.

De la primele programe scrise este bine să se țină cont de o serie de reguli privind organizarea codului sursă. Structura generală a unui cod va fi:

Apeluri ale directivelor speciale de compilare;
Declarații de biblioteci folosite;
Declarații funcții in-line;
Declarații ale prototipurilor funcțiilor personale;
Declarații variabile globale;
Corpul programului principal
{
declarații variabile;
apeluri de funcții;
}
Corpurile funcțiilor dezvoltate de programator;

Există câteva observații suplimentare privind tehnoredactarea codului sursă de care ar fi bine să se țină seama.

1. Să nu se scrie instrucțiunile una după alta ci una sub alta.
2. Toate instrucțiunile care formează un bloc pentru o altă instrucțiune să fie deplasate față de începutul respectivei instrucțiuni.

2.2. Etapele generării unui program executabil pornind de la o problemă reală

Una din greşelile începătorilor în programare constă în faptul că pun prea mult accent pe introducerea unor programe gata scrise, fără ca în prealabil să studieze problema de pornire precum şi algoritmul generat în urma analizei acestei probleme.
În momentul gândirii unui program, pe lângă alte considerente trebuie ținut cont şi de faptul că este o mare diferență între programele **bune** si cele **care merg.** Un program bun nu numai că merge, dar este şi uşor de înțeles şi menținut. Mulți dintre începătorii în

programarea C-ului au tendința de a se limita la programe care să meargă, şi acest fapt conduce la deprinderi nesănătoase.

De aceea mai jos vom prezenta paşii necesari a fi urmați în elaborarea unei aplicații pornind de la o problemă reală.

1. Înțelegerea problemei.
2. Realizarea modelului matematic, dacă este cazul.
3. Verificarea modelului matematic cu un set de date real sau calculate manual. Dacă se observă neconcordanțe se va relua verificarea punctelor anterioare, dacă nu se continuă.
4. Identificarea datelor de intrare, a datelor de ieşire precum şi a variabilelor intermediare.
5. Alegerea tipurilor de date folosite pentru reținerea în cursul procesării a respectivelor date.
6. Specificarea modului de interfațare cu utilizatorul la nivel de intrare sau ieşire.
7. Generarea schemei logice sau a pseudocodului ce descrie algoritmul propus pentru rezolvarea problemei.
8. Verificarea modelului creat la punctul 7 cu un set date reale sau calculate manual. În caz de neconcordanțe se reia verificarea punctelor anterioare, dacă nu se continuă.
9. Scrierea codului sursă (transpunerea într-un limbaj de programare oarecare).
10. Eliminarea erorilor de sintaxă.

Pentru a exemplifica paşii menționați anterior vom lua un exemplu extrem de simplu.
"Să se realizeze un program care rezovă o ecuație oarecare de gradul II"

Pasul 1
În acest caz este simplu, e clar că ne referim la o ecuație de foma $ax^2+bx+c=0$;

Pasul 2
Modelul matematic este cel studiat în liceu:
Fie $\Delta = b^2 - 4 \cdot a \cdot c$ atunci :

Dacă $\Delta>0$ atunci: $x_{1,2} = \dfrac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$.

Dacă $\Delta<0$ atunci: $x_{1,2} = \dfrac{-b \pm i \cdot \sqrt{-b^2 + 4 \cdot a \cdot c}}{2 \cdot a}$.

Dacă $\Delta=0$ atunci: $x_{1,2} = -\dfrac{b}{2 \cdot a}$.

Pasul 3
În acest caz poate fi evitat modelul matematic este prea simplu.

Pasul 4
Să vedem deci care ar fi datele ce trebuiesc furnizate de utilizator. Având în vedere că se doreşte rezolvarea unei ecuații generice de ordinul II este clar că :
- a,b,c date de intrare

- $x_{1,2}$ date de ieşire
- Δ dată intermediară.

Analizând modelul matematic observăm că a trebuit să folosim un simbol suplimentar funcție de care se face întreaga discuție. Acest simbol nu este nici dat de utilizator şi nici nu interesează pentru rezultatul final, deci Δ este o dată intermediară de calcul.

Pasul 5

Acum trebuie să specificăm tipul de dată folosit pentru reprezentarea internă (în memoria calculatorului) a variabilelor. Deci avem câteva date suplimentare la dispoziție pentru a vedea tipul de reprezentare care trebuie ales:

- specificațiile utilizatorului
- modelul matematic
- experiența în programare

În acest caz modelul matematic ne spune că a,b,c sunt numere reale. Fără o specificare clară a utilizatorului care să îndice că ele aparțin altui interval este obligatoriu ca să luăm în considerare modelul cel mai extins, deci cel mai adaptabil. Desigur că nu vom alege cel mai cuprinzător tip de date pus la dispoziție de către compilator în lipsa unor specificații externe. Deci a,b,c vor fi de tipul **float**.

Variabila intermediară Δ este rezultat al aplicării unei funcții lineare asupra unor variabile de tip real deci şi delta va trebui să fie **float**.

Datele de ieşire $x_{1,2}$, după cum se observă sunt rezultatul unei înpărțiri şi deci obligatoriu sunt reale indiferent de tipul de dată ales pentru datele de intrare.

Pasul 6

În lipsa unor specificații de la utilizator vom alege pentru citirea datelor cea mai simplă metodă citirea directă după cum am făcut şi până acum. Pentru afişare însă avem două metode afişarea de tipul Re(x) şi Im(x) sau direct rezultatul ca în calculul manual. Vom alege cea de a doua metodă.

Pasul 7

Pseudocodul va arăta ca mai jos.

```
citeşte a0
citeşte b0
citeşte c0
calculează delta după formulă delta=b*b-4*a*c;0
dacă delta >=0 atunci
{
x1=(-b+sqrt(delta))/(2*a);
x2=(-b-sqrt(delta))/(2*a);
afiafişează x1,x20
}
altfel
{
delta=-delta;
delta=sqrt(delta)/(2*a)
```

*afişează x1 de forma re+i*im*
*afişează x2 de forma re-i*im*
}

Câteva observații relative la construcția pseudocodului

- se observă că nu am tratat separat situația delta=0 pentru că zero la adunare nu modifică rezultatul
- în a doua parte am folosit un artificiu de calcul în sensul că am păstrat tot în variabila delta rezultate parțiale de calcul. Altfel ar fi trebuit să introducem variabile suplimentare pentru păstrarea părții întregi şi a părții reale. Codul ar fi fost mai clar dar în acelaşi timp am fi folosit mai multă memorie. Dacă în cazul acestui program câteva variabile suplimentare nu ne deranjează, în cazul unor programe foarte mari acest lucru poate ridica greutăți ce nu pot fi eliminate decât prin trecerea la alocarea dinamică a memoriei. Această metodă va fi prezentată ulterior.
- se observă că am scris mult cod suplimentar care din punct de vedere al calculului brut nu ne trebuie, dar din punct de vedere al modului în care îi prezint rezultatul uni utilizator este esențial pentru că este exact în forma matematică cunoscută.

Pasul 8

Va fi evitat pentru acest exemplu deoarece este prea simplu.

Pasul 9

Codul în cazul limbajului C va arăăta ca mai jos:

```
        #include <stdio.h>
*       #inlude <conio.h> // aici eroare de sintaxă detectabilă de interpretor

        void main(void)
        {
         float a,b,c,delta,x1,x2;

         clrscr()
         printf("Pentru calculul ecuatiei a*x*x+b*x*c=0 dati coeficientii");
        printf("\na=");
**      scanf("%f",a); //aici eroare de sintaxă nedetectabilă de interpretor
        printf("\nb=");
        scanf("%f",&b);
        printf("\nb=");
        scanf("%f",&b);
***     delta=b*b+4*a*c; // aici eroare de calcul
         if( delta >=0)
         {
         x1=(-b+sqrt(delta))/(2*a);
         x2=(-b-sqrt(delta))/(2*a);
         printf("\nx1=%f",x1)
         printf("\nx2=%f",x2)
                }
```

```
else
{
        delta=-delta;
delta=sqrt(delta)/(2*a)
printf("x1=%f+i*%f",-b/(2*a),delta);
printf("x2=%f-i*%f",-b/(2*a),delta);
}
}
```

Pasul 10

După cum se observă am introdus intenționat trei tipuri de greşeli. Prima, marcată cu o stea este greşeală de sintaxă ce va fi detectată de compilator şi deci o putem corecta în această fază.

Pasul 11

În cadrul rulării programului rezultat se observă că acesta nu funcționează corect. La execuția pas cu pas se va observa că în variabila a nu se depune ceea ce se citeşte de la tastatură. Este o greşeală de logică de programare, pentru că **scanf** aşteaptă un pointer pentru a returna corect rezultatul, iar a nu este declarat pointer trebuie transmisă adresa lui a. Această eroare nu poate fi detectată de interpretor.

De asemenea cu trei stele am marcat o greşeală de transpunere a algoritmului, o greşeală simplă de transcriere a unei formule de calcul. Deabia după eliminarea acestor erori se poate continua testarea programului.

Este bine ca să existe un comentariu la începutul oricărei sursei C având următorul conținut

```
/*----autor:
------data creării/modificării
------copyright
*/
```

3. Noțiunea de variabilă şi tipuri de variabile.

O variabilă este un identificator (nume) dat după dorință de către utilizator unei zone din memorie cu care se va lucra (stoca sau modifica datele utile programului) pe parcursul scrierii codului sursă.

În limbajului C pot face asemenea referiri directe la o zonă de memorie (prin intermediul pointerilor) având totodată avantajul sintaxei facile a unui limbaj de nivel mediu.

Programul sau codul sursă este tradus prin intermediul compilatorului C într-un program executabil. În cadrul acestui proces se face identificarea dintre numele variabilei şi un anumit număr de locații de memorie. Tot în sarcina compilatorului rămâne şi generarea de instrucțiuni care să se ocupe de scrierea sau citirea în bloc a mai multor locații de o dată. O variabilă poate ocupa mai mult de o locație de memorie.
Caracteristicile variabilelor sunt :

1. Fiecare variabilă are un nume.
2. Fiecare variabilă aparține unui tip de dată.
3. Fiecare variabilă reține o valoare sau un set de valori utile pentru program. Aceste valori pot fi:
   - depuse în memorie prin inițializare în program
   - pot fi citite de la tastatură sau de la alt dispozitiv de intrare.

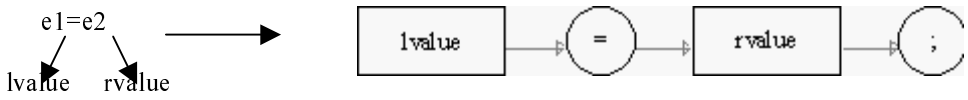
Legat de numele unei variabile se pot face câteva observații:

1. Numele unei variabile se alege de obicei astfel încât să descrie ce înseamnă pentru programator, din punct de vedere al pseudocodului într-o formă care poate fi prescurtată sau nu după preferință. De exemplu în cazul problemei din figura 1.2 numele unei variabile ce reține o notă putea fi : nota_1 sau n1 sau Nota_1. Ca observație limbajul C este “case senzitiv” (adică face distincție între literele mici şi mari folosite în scrierea codului sursă) deci NU SE CONSIDERĂ A FIIND ACELAŞI NUME DE VARIABILĂ ÎN CAZUL ÎN CARE ESTE SCRIS CU LITERE MARI SAU CU LITERE MICI. Mai concret nota_1 ≠ Nota_1≠NOTA_1.
2. Când se doreşte ca un nume de variabilă să fie compus din două cuvinte NU SE VA LĂSA SPAȚIU ÎNTRE ELE căci vor fi considerate două cuvinte separate, ele trebuiesc unite cu linie inferioară (underscore).
3. Numele de variabile suferă şi de următoarele restricții:
   – nu pot începe cu un număr (5j)
   – nu pot începe cu $ ($ abc)
   – nu pot fi cuvinte rezervate (int)
   – nu pot conține caractere speciale (bab*#*@l) excepție făcând “underscore” ex. bad_babe

3.1. Inițializarea variabilelor

În lipsa unei inițializări explicite, o variabilă locală va conține o valoare oarecare (aleatoare), numită în mod curent "garbage". Această valoare se găsea în memoria alocată respectivei variabile, la lansarea în execuția a programului

Inițializatorii expliciți se plasează după declaratorul la care se referă, separați de acesta prin semnul egal (de exemplu nr1=234).



Dacă obiectul inițializat este un agregat (un conglomerat de date recunoscut sub un nume comun), pentru inițializare se folosește o listă de expresii închisă în acolade unde la fiecare componentă corespunde câte o expresie.
Trebuie respectate următoarele reguli:

1. numărul de inițializatori nu poate fi mai mare decât numărul de componente de inițializat.
2. nu se pot inițializa decât obiecte sau tablouri cu dimensiune definită (adică se cunoaşte numărul maxim de valori de intrare ce pot fi date de utilizator pentru a fi procesate).
3. expresiile folosite pentru inițializare trebuie să fie constante în următoarele situații:
   - la inițializarea unui obiect de tip static (în C++, ne se mai aplică);
   - la inițializarea unui tablou, unei structuri sau unei uniuni.
4. nu se pot inițializa identificatori cu valabilitate în interiorul unui bloc, dar cu legare externă sau internă.
5. dacă o listă închisă între acolade are mai puține elemente decât componente are agregatul inițializat, restul componentelor se vor inițializa automat.

Obiectele de tip scalar (o serie de date de acelaşi tip, a căror definiție poate fi dată ca o înşiruire de date ce pot fi identificate prin poziția lor într-o înşiruire) se inițializează folosind o singură expresie, care poate fi eventual închisă în acolade. Uniunile (grupări de mai mulți biți de dimensiune divizibilă cu doi) se inițializează cu o singură expresie, corespunzând ca tip primului membru al uniunii.

Structurile (grupări de diverse date de tipuri diferite sub un nume comun) şi uniunile se pot inițializa cu o expresie de tip structură sau uniune corespunzătoare. Valoarea inițială a structurii sau uniunii va fi valoarea obiectului citat.

Prezența unei inițializări într-o declarație, indică o declarație definitorie (ce specifică cu ce va fi umplut respectivul spațiu de memorie). Mai jos este prezentat un exemplu.

```
int  zile [7] = { 1, 1, 1, 1, 1, 1, 1 } ;
char name1 [] = { "Costica" } ;
char name2 [] = "Dragan" ;
struct datep
    {
    int    i ; // nr. ani
    char   str [21] ; //nume
    double d ; //salar
    } vas = { 28, "Vasile", 1500000.0 } ;
```

Aceeaşi variabilă nu se poate iniţializa cu două valori, decât în cazul în care se renunţă la valoarea anterioară cu care a fost iniţializată.

Nu pot exista două declaraţii definitorii (care îi specifică conţinutul) într-un program.

Introducere in programare

3.2.Tipuri de variabile

Una din cele mai dificile probleme, care apar în cadrul învăţării unui limbaj de programare, este descrierea unei situaţii din lumea reală folosind elementele respectivului limbaj.

În afară de propunerea unui algoritm pentru a rezolva o problemă reală, programatorul mai este confruntat şi cu un alt aspect dificil: alegerea tipului de dată pentru o variabilă trebuie făcută în concordanţă cu valorile care le poate lua aceasta.

De aceea primul pas este să vedem ce tipuri fundamentale de date ne pune la dispoziţie limbajul C precum şi gama de valori în care evoluează acestea. O dată ştiute aceste lucruri putem începe să vedem care este legătura cu exemplele din viaţa reală.

Din analiza informaţiei care circulă curent între oameni rezultă că există, din punct de vedere al procesării, două tipuri de informaţie.

- informaţie obţinută prin măsurători asupra fenomenelor reale, care sunt exacte şi asupra cărora se fac prelucrări matematice de obicei;
- informaţie sub formă condensată care de obicei este recompusă asociativ în mintea umană, cum ar fi limbajul si informaţia scrisă.

De aici rezultă că într-un limbaj de programare avem nevoie de două tipuri de variabile, care să poată fi folosite pentru:

- stocarea unor valori numerice.
- stocarea unor caractere, informaţie scrisă.

Conform acestor necesităţi limbajul C a fost prevăzut cu următoarele tipuri fundamentale de date:

- **int** stochează numai numere întregi;
- **float, double** pentru numere zecimale , calcul în virgulă mobilă;
- **char** pentru a stoca codul unui caracter sau un număr întreg reprezentat pe maximum opt biţi incluzând şi semnul.

Observaţie : char[nr] va reţine un şir de nr caractere sau întregi.

Una dintre utilizările şirurilor de caractere ar fi mesajele de dialog cu utilizatorul unui program, sau mesajele de eroare. În acest caz nu ne interesează conţinutul din punct de vedere al valorii lor numerice ca atare.

Funcţie de necesităţile programatorului putem spune că există două clase de tip de date care pot fi folosite în cazul unui limbaj de nivel mediu sau înalt:

1. tipuri de date predefinite
2. tipuri de date definite de utilizator.

4. Tipuri de date predefinite

4.1. Date numerice

Limbajul C este prevăzut cu un set de date predefinite. Aceste tipuri de date sunt concretizate prin folosirea unui număr de locații de memorie, pentru stocarea, conform unui algoritm bine definit funcție de tipul de dată, a unui tip de date existente. În cazul tipului de date char, deşi a fost introdus pentru stocarea datelor de tip alfanumeric, el poate fi folosit în calcule matematice şi logice în intervalul de valori numerice pe care le poate stoca.

Pentru tipurile de date ce stochează valori numerice întregi (prescurtat int) sau , după cum am spus char, numărul este stocat în memorie direct sub forma rezultată la conversia în binar.

În cazul datelor de tip real, float, sau double la conversia directă a numărului în baza doi s-ar fi pierdut prea multe locații de memorie şi de aceea ele se păstrează sub forma ştiințifică. Această formă se referă la faptul că există o zonă de memorie care ține mantisa şi o zonă de memorie care ține exponentul. Un exemplu de reprezentare ştiințifică din punct de vedere matematic ar fi:

$$123344445455454=1.23344445455454 * 10^{14} =1.23344445455454E+14$$

unde mantisa=1,23344445455454 iar exponentul = 14.

Mai jos prezentăm o serie de tipuri de date predefinite, dimensiunile lor şi domeniul valorilor care pot fi stocate. Se observă că s-au folosit modificatori de tip. Prim modificatori de tip se înțeleg cuvinte cheie care au ca efect, la compilare, ca dimensiunea (în octeți) pe care este reprezentat un tip de date să crească (cu un factor care este 2 sau o putere a lui doi)).

char ∈ {–128, 127}	1 octet
unsigned char ∈ {0, 255}	1 octet fără semn
signed char ∈ {–128, 127}	1 octet cu semn
int ∈ {–32768, 32767 }	1 cuvânt (2 octeti)
unsigned int ∈ {0, 65535}	1 cuvânt fără semn
signed int ∈ {–32768, 32767}	1 cuvânt cu semn
short int ∈ {–32768, 32767}	1 cuvânt cu semn
unsigned short int ∈ {0, 65535}	1 cuvânt fără semn

signed short int –32768 to 32767 1 cuvânt cu semn
long int ∈ {–2147483648, 2147483647} 2 cuvinte
signed long int ∈ {–2147483648, 2147483647} 2 cuvinte cu semn
float ∈ {–3.4E–38, 3.4E+38} 2 cuvinte
double ∈ {–1.7E–308, 1.7E+308} 4 cuvinte
long double ∈ {–3.4E–4932, 1.1E+4932} 5 cuvinte

Observație: Aceste intervale sunt de obicei specifice sistemului de operare sub care lucrează compilatorul şi maşinii pe care acesta este executat. Domeniile de mai sus sunt valabile pentru compilatorul de C++ realizat de Borland.

4.1.1 Operatorul virgulă

a,b ⇔ evaluează a.evaluează b
Acest operator permite evaluarea a două sau mai multe expresii distincte in locurile in care o singură expresie este admisă a fi folosită(asta e rațiunea introducerii lui)
De ex:

```
for ( j=0,k=100 ; k-j>0 ; j++,k-- ) este chivalent cu
j=0 ; k=100;                j=0 , k=100
{                           {
…                            …
j++;          sau           j++;
k--;                        k--;
}                           }
```

Un alt exemplu funcția break_line:

```
break_line (int interval)
    { int c,j;
      for (c=getchar(),j=0 ; c!=EOF ; c=getchar (),putchar (c),j++)
      if (j % interval = = 0 ) printf( "\n" ) ;
    }
```

4.2. Date alfanumerice

În cadrul proiectării unei aplicații pot apare pe lângă cerințe legate de calcule numerice complexe şi cerințe pentru procesarea de texte sau a unor baze de date.

Totalitatea caracterelor ce pot fi afişate sau listate în cazul lucrului sub un sistem de operare clasic (UNIX sau DOS), se numesc caractere alfanumerice (alfabetice şi numerice). Fiecare caracter are o imagine pe ecran. Această imagine este stocată în memoria nevolatilă a calculatorului şi are ataşat un cod unic la care se face apel când se doreşte afişarea sau listarea ei.

Datorită necesității ca un text editat pe un sistem fabricat de o anume firmă să-şi păstreze înțelesul s-a trecut la standardizarea codării imaginilor caracterelor. Astfel a rezultat standardul ASCII folosit la ora actuală de toate calculatoarele din seria X86.

Există 255 de caractere în acest standard. Nu vom face referiri la seturile extinse de caractere specifice editoarelor de teste sau sistemelor de operare de tip Windows sau XWindows.

Pentru stocarea unui şir de caractere limbajul C foloseşte un tablou monodimensional de caractere care poate stoca codurile a maximum 255 de caractere o dată în cazul declarării statice.

Observația 1: Dacă s-a definit variabila s ca având capacitatea de stocare de 25 de caractere iar la citirea ei sunt introduse mai mult de 25, atunci numai pentru primele 25 nu există riscul pierderii lor pe parcursul execuției programului. În acest caz pot apărea erori mascate în cod.

Observația 2: În cazul în care nu am definit numărul de caractere care poate fi stocat în respectiva variabilă, funcție de modul de construcție al compilatorului, fie se alocă dimensiunea maxim posibilă, fie la prima inițializare se alocă exact cât este necesar dar nu peste maximul definit.

Observația 3: Nu există nici o legătură între valoarea numerică x şi caracterul x care apare pe ecran. De exemplu 2 ≠ '2'.

Inițializarea unei date cu o valoare utilă poate avea loc în următoarele moduri:

- la declararea ei;
- la citirea datelor de intrare;
- oriunde în program .

4.2.1. Caractere speciale de control

Pe lângă caracterele direct afişabile mai există în limbajul C o serie de caractere de control. Acestea provin din necesitatea de control a unei imprimante şi simultan au fost folosite şi la afişarea pe terminalele alfanumerice.

Secvența	Cod	Caracter
\a	07h	BEL (alert produce un semnal sonor)
\b	08h	BS (backspace)
\f	0Ch	FF (form feed mută cursor la următoarea pagină)
\n	0Ah	LF
\r	0Dh	CR (carriage return, salt la inceput de linie nouă)
\t	09h	HT (horizontal tab)
\v	0Bh	VT (vertical tab)
\\	5Ch	\ (backslash)
\'	27h	' (apostrof)
\"	22h	" (ghilimele)
\?	3Fh	? (semnul întrebării)
\ooo	ooo	Caracterul cu codul ooo în octal
\xhh	hh	Caracterul cu codul hh în hexa

Pentru justificarea introducerii acestor caractere de control se va prezenta următorul caz: era inutilă trimiterea uni număr fix de caractere identice de fiecare dată când se dorea un tabulator (un anumit număr de spații goale) la începutul unui paragraf şi atunci s-a convenit să se folosească un caracter de control, urmând ca imprimanta să-l reconvertească intern în spații goale. Anterior s-a prezentat un tabel ce conține aceste caractere.

4.3. Operațiuni simple de citire sau afişare a datelorF

Limbajul C are la dispoziție o serie de funcții pentru citirea datelor de la tastatură sau pentru afişarea lor pe ecran. Cele mai folosite dintre ele sunt grupate în două biblioteci conio.h (CONsoleInputOutput.Header) şi (StandardInputOutput.Header) stdio.h.

4.3.1. Funcții de intrare sau ieşire cu utilizatorulF

Pentru a putea discuta primele programe este necesar studierea modului în care un program poate schimba date dispozitivele standard de intrare sau ieşire (monitor, tastatură). Din acest punct de vedere există două clase de funcții:

- **uncții de intrare:** cu ajutorul cărora sistemul de calcul primeşte de la tastatură valori numerice, comenzi şi oricare alt tip de informație ce poate fi furnizată prin intermediul acelui dispozitiv.
- **uncții de ieşire:** cu ajutorul cărora sistemul de calcul livrează rezultatele (afişează) folosind un dispozitiv standard de ieşire (monitor sau imprimantă).

Pentru a citi o dată sau un şir de date de la tastatură avem o funcție specializată care poate citi orice tip predefinit de date. Funcția are prototipul prezentat mai jos

*int scanf (const char *format [,address, ...]);*

Tipul de date este specificat de către programator prin modificarea unor parametri ai funcției. Mai jos este dat un scurt exemplu:

```
#include <stdio.h>
#include <conio.h>

void main (void)
    {
    int a;
    char c;
    char b[20];
    float d;
    clrscr(); // şterge ecranul
    scanf("%d", &a); // citeşte un întreg şi-l depozitează în
    // variabila a
    scanf("%c", &c); // citeşte un caracter
    scanf("%s", b); // citeşte un şir de caractere
    scanf("%f", &d); // citeşte un număr zecimal
    getch();
    }
```

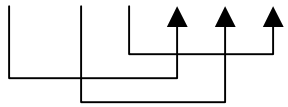
Această funcție, la apelare, preia informația introdusă de la tastatură după apăsarea tastei ENTER. Ca efect utilizatorul vede doar că programul îşi opreşte derularea şi

aşteaptă ceva, dar fără a primi un mesaj care să îi comunice ce trebuie să facă sau informații despre natura datelor care trebuiesc livrate.

Mai jos este prezentat prototipul unei funcții des folosită în afişarea diverselor tipuri de date.

*int printf (const char *format [, argument, ...]);*

Ex: printf("Afişează cele 3 numere: %d %d %d",nr1,nr2,nr3);



Vom prezenta un prim exemplu despre unele din posibilitățiе de afişare ale funcției printf .

```
#include <stdio.h>

main()
    {
    int a;          /*întreg simplu                  */
    long int b;       /* întreg lung                  */
    short int c;      /* întreg scurt                 */
    unsigned int d;     /* întreg fără semn           */
    char e;           /* caracter simplu              */
    float f;          /*număr zecimal în  simplă precizie          */
    double g;         /*număr zecimal în dubă precizie      */
      a = 1023;
      b = 2222;
      c = 123;
      d = 1234;
      e = 'X';
      f = 3.14159;
      g = 3.1415926535898;
      printf("a = %d\n",a);  /* afişare folosind baza 10 de numerație      */
      printf("a = %o\n",a);     /* afişare folosind baza 8 de numerație */
      printf("a = %x\n",a);     /* afişare folosind baza 16 de numerație */
      printf("b = %ld\n",b);    /* afişare specifică pentru întreg lung      */
      printf("c = %d\n",c);     /* afişare specifică pentru întreg scurt */
      printf("d = %u\n",d);    /* afişare specifică pentru întregfără semn */
      printf("e = %c\n",e);     /* afişare specifică pentru caracter */
      printf("f = %f\n",f);     /* afişare specifică pentru număr în simplă precizie */
      printf("g = %f\n",g);     /* afişare specifică pentru număr în dublă precizie */
      printf("\n");                    /*salt la line nouă*/
      printf("a = %d\n",a);     /* afişare specifică pentru întreg */
      printf("a = %7d\n",a);    /* afişare specifică pentru un  întreg dar alocând 7
    caractere în dreapta */
```

```
    printf("a = %-7d\n",a);   /* afişare specifică pentru un  întreg dar
                                 alocând 7 caractere în stânga */
    c = 5;
    d = 8;
    printf("a = %*d\n",c,a);   /* folosim un câmp de  5 caractere     */
    printf("a = %*d\n",d,a);   /* folosim un câmp de  8 caractere */
    printf("\n");
    printf("f = %f\n",f);  /* afişare pentru număr în simplă precizie       */
    printf("f = %12f\n",f);    /* folosim un câmp de  12 caractere */
    printf("f = %12.3f\n",f);  /* folosim un câmp de  3 caractere după
                                 virgulă */
    printf("f = %12.5f\n",f); /* folosim un câmp de  5 caractere după
                                 virgulă */
  printf("f = %-12.5f\n",f); /* afişare specifică pentru un  număr în
                                simplă precizie dar alocând 7 caractere în stânga */
  }
```

Să reluăm exemplul prezentat la citirea datelor introducând de această dată şi mesajele pentru utilizator.

```
#include <stdio.h>
#include <conio.h>

void main (void)
    {
    int a;
    char c;
    char b[20];
    float d;
    clrscr(); // şterge ecranul
    printf("Introduceti o valoare intreaga")
    scanf("%d", &a); // citeşte un întreg şi-l depozitează în //   variabila a
    printf("\nValoarea citita este %d", a);
    printf("\nDati un caracter ")
    scanf("%c", &c); // citeşte un caracter
    printf("\nCaracterul citit este %c",c);
    prinf("\nDati un sir de caractere ");
    scanf("%s", b); // citeşte un şir de caractere
    printf("\nS-a citit urmatoarea propozitie %s",b);
    printf("\nDati un numar zecimal");
    scanf("%f", &d); // citeşte un număr zecimal
    printf("\n Sa citi numarul %f",d)
    printf("\nApasa orice tasta pentru a continua")
    getch();
    }
```

5. Operatori ai limbajului C

Un operator reprezintă din punct de vedere al programării, acel simbol care alăturat la una sau două nume de variabile specifică operația ce trebuie realizată asupra lor. Funcție de numărul de variabile (deci de date) asupra căruia se aplică operatorii pot fi clasificați în două mari categorii:

- operatori unari, a căror folosire poate fi ca mai jos:
  - nume_var_i **oper_unar** nume_var_j;
  - **oper_unar** nume_var_i, unde i, j $\in$ N;

- operatori binari a căror folosire este ca mai jos:
  - nume_var_i=nume_var_j **oper_binar** nume_var_k, unde i, j, k $\in$ N.

Limbajul C are mai multe tipuri de operatori:

- operatori aritmetici care dau posibilitatea realizării oricăror calcule aritmetice;
- operatori relaționali ce sunt folosiți pentru testarea relațiilor ce pot exista între conținutul a două variabile;
- operatori logici folosiți pentru:
  - a putea realiza expresii complexe în cadrul testării relațiilor dintre variabile;
  - a procesa la nivel de bit variabilele.

5.1. Operatori aritmetici

Acest tip de operatori realizează calcule matematice simple. Desigur că precizia în care va face calculul respectiva aplicație ține de tipul cont de dată ales. Operatorii sunt prezentați în lista de mai jos. Să considerăm două variabile a şi b care la un moment dat conțin valorile a = 3 şi b = 5 atunci

- '+' adunare y=a+b→y=8
- '-' scădere y=a-b→y=-2
- '*' înmulțire y=a*b→y=15
- '/' împărțire y=a/b→y=0.6000000…0
- '%' modulo y=a%b→y=0 (restul împărțirii lui a la b)

5.2 Expresiile

O expresie constă dintr-un conglomerat realizat din:

→ operanzi
→ operatori

în scopul calculării unei valori.
Blocurile de bază constructive ale expresiilor pot fi :

– variabile
– constante

– apeluri funcție

Există patru mari tipuri importante de expresii:.

1. *Expresii constante* care rețin numai valori constante.

De exemplu 5
5+6*13/30
'a'

2. *Expresii întregi* care sunt expresii care după realizarea tuturor conversiilor automate sau explicite de tip, produc un rezultat care este de tip între. Presupunând j,k întregi atunci următoarele expresii întregi:

j
j*k
j/k+3
k-'a'
3+(int)5.0

3. *Expresii în virgulă mobilă* care sunt expresii care după toate conversiile automate sau explivite de tip, produc un rezultat care este în virgulă mobilă. Dacă x este float sau double, atunci următoarele expresii sunt de acest tip:

x
x+3
x/y*5
3.0
3.0-2
3+(float) 4

4. *Expresii de tip pointer* care sunt expresii care în urma evaluării dau o adresă. De exemplu dacă p este pointer, iar j este întreg:

p, p+1, (char*) 0x00ffffff, &j, "abc" sunt expresii de tip pointer.

5.2.1 Ordinea de evaluare a unei expresii funcție de precedența operatorilor.d

Unul din primele lucruri de care trebuie ținut seama, când se scriu expresii aritmetice complexe, este faptul că spațiile albe (spațiu, tabulator, salt la linie nouă) nu au înțeles (nu sunt luate în considerare). De exemplu cele două expresii de mai jos sunt echivalente:

```
y=x*x+2*x-5*x/y;
y=x * x + 2 * x – 5 * x / y;
```

Oamenii folosesc câteodată spațiu în loc de paranteze pentru a specifica modul de evaluare a unei expresii. Noțiunea de evaluare se referă la ordinea în care se vor realiza calculele dintr-o expresie matematică. În cadrul unui limbaj de programare numai

parantezele joacă acest rol. În lipsa parantezelor dintr-o expresie se vor respecta regulile de precedență implicită.

Ca un exemplu să discutăm modul de evaluare a expresie de mai sus dacă se va ține seama numai de precedențele implicite. Deoarece operatorii {*, /, %} au aceeaşi precedență care însă este mai mare ca a operatorilor {+,-} evaluarea expresiei se va face de la stânga la dreapta după cum urmează:

1. se calculează valoarea lui x*x;
2. se calculează valoarea lui 2*x;
3. se calculează valoarea lui 5*x;
4. se calculează valoarea lui (5*x)/y;
5. se face suma (x*x)+(2*x);
6. se face scăderea (x*x)+(2*x) - (5*x)/y.

Noțiunea de precedență se referă la modul în care iau decizii în evaluarea unei expresii dacă nu am puse paranteze. Mai jos este prezentat tabelul cu toate precedențele implicite:

Operatori	Ordinea de precedență
() [] →	de la stânga la dreapta
! ~ ++ -- + - (unary) & (type) sizeof	de la dreapta la stânga
* / %	de la stânga la dreapta
+ -	de la stânga la dreapta
<< >>	de la stânga la dreapta
< <= > >=	de la stânga la dreapta
== !=	de la stânga la dreapta
&	de la stânga la dreapta
^	de la stânga la dreapta
|	de la stânga la dreapta
&&	de la stânga la dreapta
||	de la stânga la dreapta
?:	de la dreapta la stânga
= += -= *= /= %= &= ^= |= <<= >>=	de la dreapta la stânga
,	de la stânga la dreapta

5.3. Tipuri aritmetice de date

Numerele întregi sunt reprezentate diferit de cele în virgulă flotantă. De aceea apare următoarea problemă: conversia numerelor dintr – un format într-altul.

Dacă se doreşte ca rezultatul să fie stocat într-un alt format mai mult sau mai puțin cuprinzător atunci acesta trebuie depus într–o variabilă de respectivul tip şi conversia se va face automat. Există cazuri în care compilatorul, deşi acceptă conversia, dă un mesaj

de atenționare. Această situație apare atunci când se poate altera valoarea unor date deoarece formatul în care fac conversia este mai mic.

Dacă programatorul este sigur că doreşte respectiva conversie atunci trebuie să realizeze o conversie explicită pentru ca respectivul mesaj de atenționare să dispară. Deoarece la conversiile implicite se poate altera valoarea calculată poate apărea o eroare greu detectabilă.

S1= (int)((double)z/x+0.5);

În exemplul de mai sus se forțează, trunchierea de la o reprezentare de tip double la o reprezentare în numere întregi.

Datele nu pot fi alterate dacă se realizează o conversie de tip între un tip de dată cu domeniu de valori mai mic şi un tip de dată al cărui domeniu este mai mare.

5.4. Operatorii unari ++ şi - -

După cum s-a menționat aceşti operatori se numesc unari deoarece se aplică asupra unei singure variabile (expresii) la un moment dat. Ei realizează incrementarea sau decrementarea valorii respective cu o unitate.

Limbajul C mai are un avantaj: aceşti operatori pot fi folosiți pentru post sau preincrementare. Acest lucru este prezentat în exemplu de mai jos:
Dacă x=4 atunci:

Preincrementare	y=++x ←→	x=x+1=5	şi y=x=5
Predecrementare	y=--x ←→	x=x-1=3 şi y=x=3	
Postincrementare	y=x++ ←→	y=x=4	şi x=x+1=5
Postdecrementare	y=x-- ←→	y=x=4	şi x=x-1=3

5.5. Operatorii op=

Au fot introduşi pentru simplificarea scrierii codului. O instrucțiune generică de tipul rez = rez op opd1 ←→ rez op = opd1. Aceşti operatori aparțin următoarei clase { += -= = /= %= &= ^= |=} de exemplu: x=x*7 ←→x*=7;

5.6. Operatori relaționali şi logicid

Pentru a se putea lua decizii într-un program, deci pentru a realiza ramificarea fluxului de prelucrare a datelor, trebuie să existe o modalitate de comparare a informațiilor şi de asemeni de combinare a expresiilor logice primitiv în expresii mai complexe.

De exemplu dacă în variabilele x şi y am stocată valoarea 4 atunci expresia x==y va fi ADEVĂRATĂ şi va avea la evaluarea expresiei valoarea numerică 1, dacă x==4 şi y=5 atunci expresia x==y va fi FALSĂ şi deci în urma evaluării va fi înlocuită cu valoarea 0.

Operatori relaționali	Explicație
==	Egal cu
!=	Diferit de
>	Mai mare ca
<	Mai mic ca
>=	Mai mare sau egal
<=	Mai mic sau egal
Operatori logici	
||	SAU
&&	ŞI
!	NU

5.7. Expresii logice complexe

De obicei este insuficientă o singură expresie logică simplă. În scrierea unui program este necesară de multe ori compunerea a mai multe decizii. Un exemplu clasic este cazul în care trebuie verificată aparteneța valorii stocate într-o variabilă la un anumit interval. De exemplu $x \in [5,20]$ este echivalent cu $x \geq 5$ şi $x \leq 20$ ceea ce folosind operatori logici şi relaționali se poate scrie: (x>=5)&&(x<=20).

Tabelele de adevăr pentru operatorii logici ai limbajului sunt:

X	Y	X&&Y	X||Y	!X
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Urmează prezentarea un program simplu care realizează diverse operații logice cu un set de valori inițializate direct în program. Această metodă de inițializarea are dezavantajul că programul nu poate primi pentru calcul alte valori decât dacă se modifică codul sursă.

```
void main(void)
    {
    char mask;
    char number[6];// tablou de 6 elemente de tip char
    char and,or,xor,inv,index;
      number[0] = 0x00;// inițializare în cod
      number[1] = 0x11;
      number[2] = 0x22;
      number[3] = 0x44;
      number[4] = 0x88;
      number[5] = 0xFF;
      printf(" numerele asupra carora am aplicat diverse operatii logice folosind o
    masca sunt: \n");
```

```
    mask = 0x0F;
    for (index = 0;index <= 5;index++) {
      and = mask & number[index];// şi logic
      or = mask | number[index];//sau logic
      xor = mask ^ number[index];// sau exclusiv
      inv = ~number[index];// not bit cu bit
      printf("%5x %5x %5x %5x %5x %5x\n",number[index],
            mask,and,or,xor,inv);
    }

    printf("\n");
    mask = 0x22;
    for (index = 0;index <= 5;index++) {
      and = mask & number[index];
      or = mask | number[index];
      xor = mask ^ number[index];
      inv = ~number[index];
      printf("%5x %5x %5x %5x %5x %5x\n",number[index],
            mask,and,or,xor,inv);
    }
  }
```

Pentru restul tipurilor de operatori prezentați vom da un alt exemplu de folosire a lor.

```
void main(void)
  {
  int x = 0,y = 2,z = 1025;
  float a = 0.0,b = 3.14159,c = -37.234;
                                  /* incrementare */
    x = x + 1;      /*  x se incrementează */
    x++;            /* x se postincrementează */
    ++x;            /* x se preincrementează x */
    z = y++;        /* z = 2, y = 3 */
    z = ++y;        /* z = 4, y = 4 */

                                  /* decrementare */
    y = y - 1;      /* y se decrementează */
    y--;            /* y este postdecrementat */
    --y;            /* y este predecrementat */
    y = 3;
    z = y--;        /* z = 3, y = 2 */
    z = --y;        /* z = 1, y = 1 */

                                  /* operații aritmetice */
    a = a + 12;     /* se adună 12 la a */
    a += 12;        /* se adună încă 12 la a */
```

```
  a *= 3.2;       /* a se înmulțește cu 3.2 */
  a -= b;         /* se scade b din a */
  a /= 10.0;      /* se împarte a la 10.0 */

                  /* expresii condiționale */
  a = (b >= 3.0 ? 2.0 : 10.5 );     /* această expresie    */

  if (b >= 3.0)                 /* este identică cu toată instrucțiunea if… */
    a = 2.0;                       /* din punct de vedere al rezultatului*/
else
    a = 10.5;

  c = (a > b?a:b);        /* c primeşte maximum dintre a şi b */
  c = (a > b?b:a);        /* c primeşte minimul dintre a şi b */
}
```

6. Instrucțiuni de selecție (decizie).S

6.1. Instrucțiune simplă de decizie

Fie următoarea problemă ce trebuie rezolvată prin intermediul unui program:

ă se realizeze un program care face automat calculul mediei de licență şi a medieiS generale.

Programul primeşte trei medii de le examenele de licență şi nota primită la susținerea proiectului de stat. El trebuie să calculeze media între notele de la examenele de licență, rezultând o primă medie, care apoi se mediază cu nota de la proiectul de stat rezultând media generală.

Există o limitare asupra datelor de intrare care trebuie realizată de program şi anume dacă una din mediile de la examenul de licență este mai mică ca cinci să nu mai realizeze calculul mediilor finale şi să-l declare pe respectivul candidat respins.

O soluție de rezolvare a problemei la nivel de pseudocod arată ca mai jos:

```
include librăriile necesare lucrului în mod textS
void main(void)
    {//început program
    declară variabilele n1,n2,n3,ns,m1,m2 de tip real;0
    şterge ecranul;0
    citeşte nota1 în variabila n1;0
    citeşte nota2 în variabila n2;0
    citeşte nota3 în variabila n3;0
    dacă (n1>5) şi (n2>5) şi (n3>5) atunciS
                {
            citeşte nota stat în variabila ns;
                realizează media m1=(n1+n2+n3)/3;S
                realizează media m2=(m3+ns)/2;S
                afişează media de licența m1;S
                afişează media generală m2;S
                }
    altfel afişează "Candidat respins";0
    aşteaptă o tastă;S      }// sfârşit de program0
```

Se observă că a fost nevoie de o ramificare la nivelul procesării datelor, ramificare care apare din nevoile utilizatorului. Este nevoie ca la un moment dat să hotărâm moduri multiple de prelucrare pentru acelaşi flux de date de intrare.

În limbajul C pentru a implementa acest tip de decizii a fost introdusă instrucțiunea if… cu sintaxa ca mai jos:

if (condiție=adevărată) S *dacă condiția este adevărată*
{ *execută bloc S instrucțiuni1S*
execută bloc instrucțiuni1S altfel
} *execută bloc instrucțiuni1S*
else
{
execută bloc instrucțiuni2S
};

Sintaxa instrucțiunii este identică cu pseudocodul, bineînțeles că se foloseşte limba engleză.

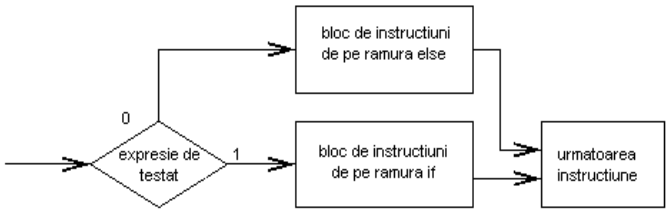


Fig. 6.1a. Reprezentarea prin schema logică a instrucțiunii de decizie0

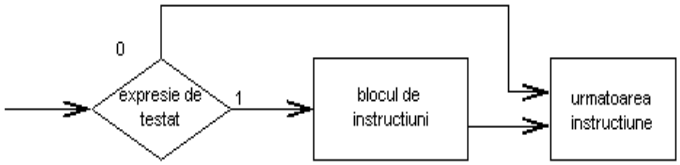


Fig. 6.1b. Reprezentarea prin schema logică a instrucțiunii de decizie simple

În figurile 6.1a şi 6.1b este prezentată reprezentarea la nivel de schemă logică a instrucțiunii de decizie. Spre deosebire de alte limbaje de programare, cum ar fi PASCAL de exemplu, limbajul C nu foloseşte variabile de tip logic pentru rezultatul evaluării unei expresii. Valorile comparate pot rezulta din valoarea returnată de o funcție, din cea stocată într-o variabilă sau direct dată în program.

Dacă în locul condiției, în urma evaluării expresiei rezultă un număr diferit de zero, acesta va fi considerat ca condiție îndeplinită şi deci se va executa blocul1 de instrucțiuni, dacă este zero condiția nu este satisfăcută atunci se va executa blocul2 de instrucțiuni. Acest lucru conferă un mare avantaj prin faptul că se pot introduce direct expresii matematice sau apeluri de funcții ce returnează rezultate numerice în zona de test a condiției.

Nu întotdeauna este nevoie de ambele părți ale instrucțiunii, de exemplu când se execută o operațiune numai dacă este îndeplinită o anumită condiție. De fapt structura logică de control ar trebui prevăzută şi cu o ramură de else dar acest lucru nu ar duce decât la supraîncărcarea codului, ceea ce îngreunează mult depanarea programului, rezultând sintaxa prezentată mai jos:

if (condiție=adevărată)S *dacă condiția este adevărată atunciS*
{ *execută bloc S instrucțiuniS*
execută bloc instrucțiuniS
};

În cazul în care sunt implicate mai multe decizii simple care trebuiesc luate funcție de diferite valori ale unei singure variabile, limbajul este prevăzut cu instrucțiune specializată case…. Discuția acestei instrucțiuni se va face ulterior.

Mai jos este prezentat programul corespunzător pseudocodului prezentat anterior.

```
#include <stdio.h>
#include <conio.h>

void main(void)
        {
        float n1,n2,n3,ns,m1,m2;

        clrscr();
        printf("Dati prima nota:");
        scanf("%f",&n1);
        printf("\nDati a doua  nota:");
        scanf("%f",&n2);
        printf("\nDati prima nota:");
        scanf("%f",&n1);
        if( (n1>5) && (n2>5) && (n3>5))
                {
                printf("\nDati  nota de la stat:");
                scanf("%f",&ns);
                m1=(n1+n2+n3)/3;
                m2=(m3+ns)/2;
                printf(" media de licenta %f", m1);
                printf(" media generala  %f",m2);
                }
        else
        printf("Candidat respins");
        getch();
        }
```

Observații:S

1. În această fază faptul că s-a folosit o referință, pentru citirea unei variabile (vezi &n1) nu poate fi explicat suficient şi va trebui luat ca atare în cazul citirii variabilelor simple.
2. Operatorul '\n' realizează un salt la linie nouă.

3. Se observă că pe ramura de else nu s-au introdus delimitatorii de bloc (acoladele) pentru că dacă aceştia nu sunt precizați prima instrucțiune este automat considerată aparținând ramurii respective.
4. Pentru programatorii mai avansați: în momentul în care blocul de instrucțiuni este format dintr-o singură instrucțiune se poate folosi operatorul ? ca în exemplul de mai jos, unde :

```
if(a>b) x=25;
else yx=34;
```

poate fi echivalat prin: *a>b (x=25) : (y=34);*

Un alt exemplu de folosire a operatorului ? este o linie de cod care calculează maximul dintre var1 şi var2 şi-l depune în variabila max.

```
max=(var1>var2) ? var1 : var2;
```

Mai jos vom prezenta un program simplu cu exemple despre evaluarea unor expresii compuse într-un program:

```
void main(void)
{
int x = 11,y = 11,z = 11;
char a = 40,b = 40,c = 40;
float r = 12.987,s = 12.987,t = 12.987;
  if (x == y) z = -13;   /* execuția instrucțiunii va avea ca efect z = -13 */0
  if (x > z)  a = 'A';   /* execuția instrucțiunii va avea ca efect a = 65  */0
  if (!(x > z)) a = 'B'; /* execuția instrucțiunii nu va avea nici un efect */0
  if (b <= c) r = 0.0;   /* execuția instrucțiunii va avea ca efect r = 0.0 */0
  if (r != s) t = c/2;   /* execuția instrucțiunii va avea ca efect t = 20  */0

  if (x = (r != s)) z = 1000; /* execuția instrucțiunii va avea ca efect0
                        faptul că x va lua valori pozitive iar z = 1000 */0
  if (x = y) z = 222;   /* execuția instrucțiunii va avea ca efect0
                    x = y şi z = 222 */0
  if (x != 0) z = 333;  /* execuția instrucțiunii va avea ca efect z = 333 */0
  if (x) z = 444;       /* execuția instrucțiunii va avea ca efect z = 444 */0

  x = y = z = 77;
  if ((x == y) && (x == 77)) z = 33; /* execuția instrucțiunii va avea
                                  ca efect z = 33 */
  if ((x > y) || (z > 12))   z = 22; /* execuția instrucțiunii va aveaS
                              ca efect z = 22 */
  if (x && y && z) z = 11;          /* execuția instrucțiunii va aveaS
                              ca efect z = 11 */
 if ((x = 1) && (y = 2) && (z = 3)) r = 12.00; /* execuția instrucțiuniiS
```

```
                    va avea ca efect    x = 1, y = 2, z = 3, r = 12.00 */
if ((x == 2) && (y = 3) && (z = 4)) r = 14.56; /* execuția instrucțiunii nuS
                                va avea nici un efect */

  if (x == x); z = 27.345;  /* z se va modifica întotdeauna  */
  if (x != x)  z = 27.345;  /* y nu se va modifica nici o dată */0
  if (x = 0)   z = 27.345;  /* x va lua valoarea 0 , z va rămâne neschimbat */0
}
```

6.2. Instrucțiune de decizie multiplă (selecție)S

Există situații în care rezolvarea unei probleme impune ca o decizie să conducă la execuția a mai mult de două ramuri de program. Ca exemplu să considerăm cazul celei mai simple forme de meniu care poate fi oferită unui utilizator.

Se citeşte un număr de la tastatură. Funcție de valoarea dată se poate alege execuția uneia dintre opțiunile prezentate utilizatorului. Mai jos este prezentat un meniu simplu folosind metoda pseudocodului.

```
şterge ecranS
afişează mesaj "Apasă una din taste pentru a alege operațiunea dorită"S
afişează mesaj "1- Citire din fişier"S
afişează mesaj "2- Scriere in fişier"S
afişează mesaj "3- Procesare date"S
afişează mesaj "4- afişare grafic rezultat"S
citeşte o tastă
funcție de tastă citită
        dacă e 1 :  apelează funcția de citire din fişier apoi stopS
        dacă e 2 : apelează funcția de scriere in fişier apoi stopS
        dacă e 3 : apelează funcția de procesare date apoi stopS
        dacă e 4 : apelează funcția de afişare grafică a rezultate apoi stopS
        dacă nu e nici unul din numerele de mai sus :S
                afişează mesaj "Opțiune invalidă"S
```

Se observă că se poate realiza implementarea şi cu instrucțiuni if… dar, acest lucru nu este justificat atât timp cât avem o instrucțiune specializată pentru acest tip de situații. Aceasta este instrucțiunea case… cu următoarea sintaxă:

```
switch(operand) //funcție de operandS
    {
    case val1: // dacă operandul ia valoarea 10
                bloc de instrucțiuni 10
        break; // stop
    case val2: // dacă operandul ia valoarea 20
                bloc de instrucțiuni 20
        break; // stop
```

```
                    ...
        case valn: // dacă operandul ia valoarea n0
                    bloc de instrucțiuni n0
                break; // stop
        default : // pentru orice altă valoare0
                bloc de instrucțiuni ce tratează restul valorilor0
                ce le poate avea operandul
        } // stop
```

O formă de reprezentare la nivel de schema logică ar fi cea prezentată în figura 6.2.

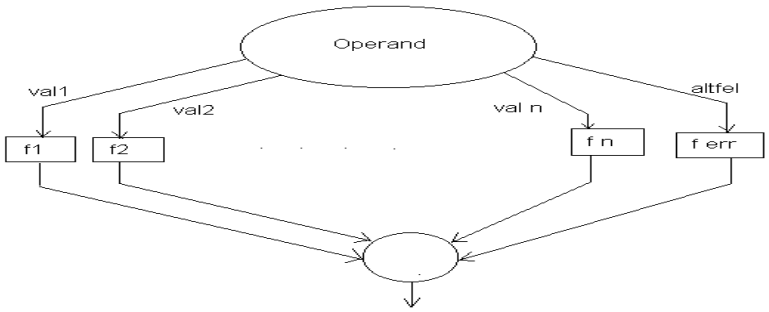


Fig. 6.2. Schema logică pentru reprezentarea unei structuriS de selecție multiplă

Mai jos este prezentat un program care citeşte doi operanzi de la tastatură apoi ne permite selecția operației matematice ce trebuie realizată ca în final să afişeze rezultatul.

```
#include <stdio.h>
#include <conio.h>

void main(void)
        {
         int op1, op2;
         float rez;
         char oper;

    clrscr();
        printf("Dati primul operand:");
        scanf("%d",&op1);
        printf("\nDati al doilea operand:");
        scanf("%d",&op2);
        printf("\nSelectionati operatia dorita");
        oper=getch();
        printf("/n1- Adunare");
        printf("/n2- Scadere");
        printf("/n3- Inmultire");
        printf("/n4-Impartire ");
         switch(oper)
```

```
        {
        case ' 1':rez=op1+op2;break;
        case ' 2':rez=op1-op2; break;
        case ' 3':rez=op1*op2; break;
        case ' 4':rez=op1/op2; break;
        default:
                clrscr();
                printf("Operatie nepermisa");
        }
        printf("\nRezultatul este: %d", rez);
}
```

7. Instrucțiuni de ciclare. Introducere

S-a discutat până acum despre structuri de control care permit alegerea condiționată între execuția mai multor seturi de instrucțiuni. Sistemele de calcul au avantajul că pot repeta, o dată "instruite ce şi cât" (programate), un număr de operațiuni de câte ori este nevoie. Orice limbaj de programare este are implementată această posibilitate. Există două clase de instrucțiuni de ciclare (repetare).

O primă categorie se referă la instrucțiunile care repetă un bloc de instrucțiuni până când este îndeplinită o condiție (prin condiție putem înțelege o expresie multiplă ce va fi evaluată de către compilator). De obicei este folosită pentru interacțiuni repetitive cu utilizatorul sau în calculul numeric prin aproximații şi de aceea instrucțiunile respective se numesc instrucțiuni cu număr necunoscut de paşi.

O a doua categorie de instrucțiuni repetitive este aceea a instrucțiunilor cu număr cunoscut de paşi. Din punct de vedere al limbajului nu ar fi strict nevoie de acest tip de instrucțiuni, pentru că se poate realiza acest lucru cu instrucțiunile din prima clasă. Totuşi, după cum se va observa mai jos implementarea (cu instrucțiuni cu număr necunoscut de paşi) pentru această a doua clasă presupune un cod care va fi identic la absolut toate interpretările. Pentru a uşura scrierea codului şi a-l face mai clar s-a implementat o instrucțiune nouă for**... .**

7.1. Instrucțiuni cu număr necunoscut de paşiS

7.1.1. Instrucțiunea while

Este o instrucțiune care permite repetarea sau nu a execuției unei bucăți de cod funcție de valoarea de adevăr a unei expresii.

Un exemplu în care ar fi necesar acest tip de instrucțiuni ar fi restricția domeniului de valori în cazul datelor care le poate introduce un utilizator. Acest lucru este necesar pentru că de obicei utilizatorul nu ține cont de toate restricțiile relativ la intervalul în care poate da o valoare de intrare,sau poate da din greşeală o valoare incorectă. Totuşi modelul matematic specifică clar că algoritmul lucrează corect numai în intervalul [a,b].

Pentru utilizator faptul că programul dă un rezultat eronat datorită unei valori de intrare introdusă greşit înseamnă că programul nu funcționează corect. Este necesar ca programatorul să nu permită introducerea altor date decât cele specificate de modelul matematic.

Observație: limbajul C spre deosebire de limbajul Pascal nu generează un mesaj de eroare dacă tipul de dată introdus nu coincide cu specificatorul dat funcției *scanf*. Din acest motiv în momentul în care se pregăteşte o versiune beta (prima versiune a aplicației care va fi folosită de către utilizator) trebuiesc introduse toate restricțiile de tipul menționat mai sus. La nivel de pseudocod metoda poate fi descrisă ca mai jos:

citeşte valoarea de intrare
atâta timp cât valoarea de intrare nu se află în intervalul [a,b]
emite un mesaj de atenționare către utilizatorS

reciteşte valoareaS

Pentru aceasta se poate folosi instrucțiunea while… a cărei sintaxă este prezentată mai jos:

```
while(conditie)// atâta timp cât condiția este adevărată execută
{
bloc de instrucțiuniS
}
```

În figura 7.1. este prezentată schema logică corespunzătoare instrucțiunii while.

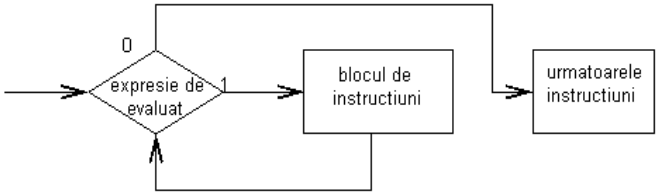


Fig. 7.1. Schema logică aS instrucțiunii while

În exemplul de mai jos este vorba de citirea unei valori întregi care trebuie să aparțină intervalului [0,255].

```
#include <stdio.h>
#include <conio.h>

void main(void)
        {
         unsigned char value;
         clrscr();
        printf("Nr1:");
        scanf("%d",&op1);
        while((value>255)&&(value<0))
        {
        printf("Eroare ! Numarul trebuie sa fie intre 0 si 255 ");
         printf("Nr1:");
        scanf("%d",&op1);
        }
        // urmează restul programuluiS
        }
```

7.1.2. Instrucțiunea do… while

După cum s-a menționat anterior instrucțiunea while… testează mai întâi valoarea de adevăr a expresiei de evaluat şi apoi, funcție de aceasta, execută sau nu blocul de instrucțiuni. De aceea i se mai spune instrucțiune de ciclare cu test anterior.

Dacă se analizează exemplul de mai sus din punct de vedere al simplității codului se observă că blocul de instrucțiuni pentru citirea variabilei a fost scris de două ori. Pentru a evita scriere de cod inutil în această situație pseudocodul trebuie să fie ca mai jos:

citeşte valoareaS
verifică dacă se află în intervalul doritS
dacă da continuă programulS
dacă nu reia de la citirea variabilei

Pentru a putea rezolva elegant situațiile de acest gen s-a mai introdus o instrucțiune şi anume do…while a cărei sintaxă este prezentată mai jos:

```
do //execută
 {
bloc de instrucțiuni0
 }while(condiție)// atâta timp cât condiția este adevărată
```

În figura 7.2. avem prezentată schema logică pentru instrucțiunea do…while.

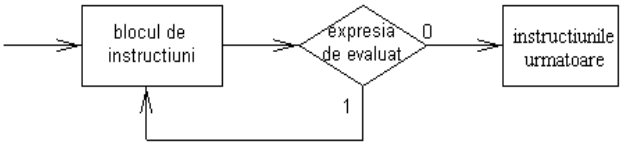


Fig. 7.2. Schema logică a instrucțiunii do…while

Folosind această instrucțiune codul prezentat mai sus devine:

```
#include <stdio.h>
#include <conio.h>

void main(void)
    {
        unsigned char value;
        clrscr();

    do
    {
        printf("Nr1:");
    scanf("%d",&op1);
    if( value>255)&&(value<0))
                printf("Eroare ! Numarul trebuie sa fie intre 0 si 255 ");
        } while((value>255)&&(value<0));

    // urmează restul programuluiS
    }
```

Deci se citeşte numărul şi, dacă acesta nu este corect, se reia citirea. Pentru validarea afişării mesajului de eroare s-a prevăzut verificarea condiției pentru că dacă utilizatorul introduce corect valoarea de prima oară, nu este permisă afişarea mesajului de eroare.

Această instrucțiune execută măcar o dată blocul de instrucțiuni şi de-abia apoi face evaluarea expresiei pentru a putea vedea dacă acesta se mai repetă sau nu. De aceea i se mai spune şi instrucțiune de ciclare cu test posterior.

7.2. Instrucțiuni cu număr cunoscut de paşi. Instrucțiunea for.S

Din cele prezentate până acum rezultă că se poate repeta un bloc de instrucțiuni până când este îndeplinită o condiție. Există cazuri când se cunoaşte de la început câte valori se doresc a fi prelucrate, fie prin specificarea directă a utilizatorului, fie ca rezultat a unor calcule anterioare. De obicei acest lucru se aplică la calculul vectorial sau matricial dar nu numai.

Întrucât nu s-a discutat încă despre acest tip de date se va lua în discuție un exemplu mai simplu, cum ar fi realizarea unui program ce va ridica un număr la o putere. Pseudocodul şi apoi programul sunt prezentate mai jos:

citeşte numărul (nr)
citeşte puterea la care va fi ridicat (pw)
inițializează variabila rezultat, ce va păstra nr la putere, cu 1 (rez=1)
atâta timp cât variabila ce a primit puterea >0 fă
*rez=rez*nr;*
pw=pw-1;
repetă
afişează rezultatul0

Codul sursă corespunzător .

```
#include <stdio.h>
#include <conio.h>

void main(void)
        {
         int nr, pw;
         float rez;

         clrscr();
        printf("Numarul");
        scanf("%d",&nr);
        printf("\nPuterea");
        scanf("%d",&op2);
        rez=1; //****
        while(pw>0)//*****
        {
        rez=rez*nr;
```

```
pw=pw-1;//*****
}
printf("Numarul %d la puterea %d este %f",nr,pw,rez);
getch();
}
```

Instrucțiunile marcate cu asterix se repetă în foarte multe situații când numărul de paşi este cunoscut. De aceea s-a implementat instrucțiunea for… având următoarea sintaxă

```
for(contor=valoareinițială;contor<sau>valoarefinală;
                                        contor=contor+sau -pas)
    {
    bloc de instrucțiuniS
    }
```

La nivel de schema logică execuția instrucțiunii for este prezentată în figura 7.3.

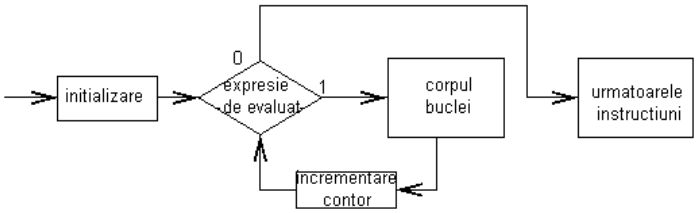


Fig. 7.3. Schema logică instrucțiunii ***for***

Acum se observă că această instrucțiune scuteşte programatorul de a scrie cod suplimentar. Programul anterior rescris folosind această instrucțiune este dat mai jos.

```
#include <stdio.h>
#include <conio.h>

void main(void)
    {
  int nr, pw,i;
  float rez;

  clrscr();
    printf("Numarul");
    scanf("%d",&nr);
    printf("\nPuterea");
    scanf("%d",&op2);
     for(i=0;i<pw;i=i+1)
                rez=rez*nr;
    //codul mai poate fi scris şi ca in zona comentata ce urmează
    // for(i=pw;i>0;i=i-1)
    //          rez=rez*nr;
    printf("Numarul %d la puterea %d este %f",nr,pw,rez);
```

```
getch();
}
```

Există un control total asupra desfăşurării buclei exact ca şi în cazul while dar codul scris este mult mai succint.
Observația 1: Dacă blocul de instrucțiuni constă într-o singură instrucțiune parantezele nu mai sunt necesare
Observația 2: Dacă se pune ";" după for acesta va considera că are o buclă vidă, compilatorul nu va raporta eroare sintactică, dar ceea ce este scris în blocul de instrucțiuni va fi doar o singură dată executat.

7.3. Alte instrucțiuniS

7.3.1. Instrucțiunea break

Efectul instrucțiunii break constă în părăsirea imediată a celei mai apropiate (textual) instrucțiuni switch, while, do, for înconjurătoare.

7.3.2. Instrucțiunea continue

Instrucțiunea continue nu se poate utiliza decât într-un ciclu while, do sau for. Efectul instrucțiunii continue constă :

- într-un salt la evaluarea condiției de ciclare, dacă cea mai apropiată (textual) instrucțiune de ciclare înconjurătoare este while sau do.
- într-un salt la evaluarea expresiei din a treia poziție , dacă cea mai apropiată (textual) instrucțiune de ciclare înconjurătoare este for.

Prezentăm mai jos un exemplu simplu de folosire a instrucțiunii break şi continue:

```
void main(void)
    {
    int xx;

      for(xx = 5;xx < 15;xx = xx + 1){
        if (xx == 8)
          break;
        printf("In bucla oprita fortat , xx are valoarea %d\n",xx);
      }

      for(xx = 5;xx < 15;xx = xx + 1){
        if (xx == 8)
          continue;
        printf("Prin aplicarea continue in bucla, xx devine %d\n",xx);
      }
    }
```

7.3.3. Instrucțiunea goto nume_etichetă

Efectul instrucțiunii goto constă într-un salt necondiționat la instrucțiunea cu eticheta citată în instrucțiunea de salt, care trebuie să se afle în aceeaşi funcție.

7.3.4. Instrucțiunea return

Efectul instrucțiunii return conduce la terminarea imediată a execuției funcției curente şi la revenirea la funcția apelantă. Dacă expresia este prezentă, ea trebuie să fie de un tip care poate fi convertit la tipul funcției în care apare instrucțiunea return. Valoarea expresiei este valoarea întoarsă de funcție.

7.3.5. Constante simbolice

Cu ajutorul directivei "#define" se pot defini la începutul programului nume sau constante simbolice care sunt un şir particular de caractere. După aceea, compilatorul va înlocui toate aparițiile, nepuse între ghilimele, ale numelui, prin şirul corespunzător. Înlocuirea efectivă a numelui poate fi orice text ea nu se limitează la numere.

```
#define MIN 0     //valoarea minimă admisă
#define MAX 300   //valoarea maximă admisă
# define AND &&

void main(void)
     {
      int x;
    printf("Dati un numar");
    scanf("%d",&x)
    if((x>=MIN)AND(x<=MAX))printf("\nNumarul este in interval");
        else printf("Numarul nu este in interval");
    getch();
    }
```

Cantitățile MIN şi MAX sunt constante, aşa încât ele nu apar în declarații. Numele simbolice se scriu în mod normal cu litere mari, aşa că pot fi uşor distinse de numele de variabile care se scriu cu litere mici. La sfârşitul unei definiții NU se pune punct şi virgulă.

8. Masive de date. Tablouri

În majoritatea calculelor ştiinţifice se foloseşte noţiunea de vector, matrice, lucrul în spaţii n-dimensionale. Limbajul C a fost prevăzut are posibilitatea de a lucra cu un masive de date de acelaşi tip, masive ce pot fi recunoscute sub un nume unic. Li se mai spune şi tablouri de date. Există posibilitatea definirii statice de tablouri cu maximum trei dimensiuni. Declararea unui tablou se realizează ca mai jos:

- tablou monodimensional:
  tip-dată nume-tablou[nr-elemente]; (ex: char s[10])
- tablou bidimensional:
  tip-dată nume-tablou[nr-linii][nr-coloane]; (ex: int m[20][20])
- tablou tridimensional:
  tip-dată nume-tablou[coord x][coord y][coord z]; (ex: float punct[10][10][10]).

Aceste declaraţii au ca efect rezervarea unui spaţiu de memorie pentru depozitarea 10 caractere sau 100 de întregi sau 1000 de valori reale pentru cazurile prezentate ca exemplu. Accesul la fiecare zonă a tabloului se face poziţional. Elementele sunt depuse unul după altul şi atunci referirea se face printr-un indice care specifică poziţia elementului respectiv (al câtelea este în şir). Deci pot exista referiri de genul:

s[3]='2' sau m[2][2]=30 sau punct[0][0][0]=0.233445.

Indicele poate fi furnizat prin intermediul unei variabile. Acest lucru permite folosirea unor instrucţiuni ciclice pentru prelucrarea datelor din cadrul unui tablou de date. O situaţie mai specială este în cazul tablourilor monodimensionale (şirurilor) de caractere. Cum, în general, prelucrarea acestora nu se face caracter cu caracter, ele fiind de obicei folosite în manipularea informaţiei scrise, limbajul C a fost prevăzut cu o serie de funcţii specializate în prelucrarea globală a lor.

8.1. Tablouri monodimensionale

După cum am spus tablourile monodimensionale pot fi folosite şi pentru calcule matematice în spaţii n-dimensionale, unde se lucrează cu un set de n coordonate. Vom prezenta câteva implementări ale operaţiunilor simple cu vectori.

8.1.1. Prelucrări simple cu vectoriS

În cazul declarării unui vector se rezervă un spaţiu pentru numărul maxim de elemente care s-ar putea să aibă nevoie utilizatorul programului. De exemplu cazul unui program specializat pentru calcule asupra unui număr de măsurători, utilizatorul poată să facă într-un caz 10 măsurători (o estimare mai grosieră) şi în altul 100. Deci se va defini un vector cu 100 de elemente dar în acelaşi timp trebuie ca programul să ceară specificarea numărului de valori ce trebuie introduse şi să facă procesarea numai asupra lor. Va trebui să folosim o instrucţiune for… deoarece are numărul de paşi cunoscut. În cazul în care se poate specifica nici o limită pentru numărul de valori ce trebuiesc stocate sau prelucrate nu se mai pot folosi tablouri definite static ci liste înlănţuite. Mai jos prezentăm un program care

citeşte doi vectori de aceeaşi dimensiune de la tastatură şi realizează o serie de operațiuni simple cu ei afişând de fiecare dată rezultatele.

Introducere în programare

```
#include <stdio.h>
#include <conio.h>

void main (void)
    {
    int v1[100], v[100],v2[100],i,j,n;
    clrscr();
    printf("Dati dimensiunea vectorilor n=");
    scanf("%d",&n);

    for(i=0;i<n;i++)
        {
        printf("V[%d]=",i);
        scanf("%d",v[i])        }//citirea vectorului v
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;i++)
         if(v[i]>v[j])
                {
                t=v[i];
                v[i]=v[j];
                v[j]=t;
                }
    printf("\nVectorul ordonat:\n")
    for(i=0;i<n;i++) printf("%d ",v[i]);// afisarea vectorului v
    for(i=0;i<n;i++)
        {
        printf("V1[%d]=",i);
        scanf("%d",v1[i]);
        }
    for(i=0;j=0;i<n;i++;j++)
        {
        v2[j]=v1[2*i];
        v2[j+1]=v[2*i+1];
        }
    printf("\nVectorul v interclasat cu v1 este:\n")
    for(i=0;i<n;i++) printf("%d ",v[i]);
    for(i=0;i<n;i++) v2[i]=v1[i]+v[i];
    printf("\nVectorul v sumat cu v1 este:\n")
    for(i=0;i<n;i++) printf("%d ",v[i]);
    getch();
    }
```

Mai jos este prezentat codul care implementează algoritmul "merge sort", bazat pe tehnica "împarte şi stăpâneşte", într-o ,manieră recursivă.

```
#include <stdio.h>

int a[100],n; // variabile globale vizibile şi în main şi în interiorul
              //oricărei funcții definite de utilizator
```

```
    merge_sort(int st,int dr); // prototipurile funcţiilor proprii
    merge(int st,int mij,int dr);
    void main(void)
        {
         int i;
         printf("\n introd dimensiunea sirului de sortat :  ");
         scanf("%d",&n);
         for(i=0;i<n;i++)
         {
         printf("\nV[%d]=",i);
             scanf("%d",&a[i]);
             }
         merge_sort(0,n-1);
         printf("\n sirul ordonat este :\n");
         for(i=0;i<n;i++)
            printf("%3d  ",a[i]);
        }

merge_sort(int st,int dr)
        {
         int mij;
         if(st<dr){
         mij=(st+dr)/2;
         merge_sort(st,mij);
         merge_sort(mij+1,dr);
         merge(st,mij,dr);
         }
        }

merge(int st,int mij,int dr)
        {
         int h,i,j,k;
         int b[100];
         h=st;
         i=st;
         j=mij+1;
         while(h<=mij && j<=dr){
         if(a[h]<=a[j]) {b[i]=a[h];h++;}
         else {b[i]=a[j];j++;}
         i++;
         }
         if(h>mij)
         for(k=j;k<=dr;k++)
                 {b[i]=a[k];i++;}
         else for(k=h;k<=mij;k++)
                 {b[i]=a[k];i++;}
         for(k=st;k<=dr;k++) a[k]=b[k];
```

```
}
```

8.1.2. Şiruri de caractere

S-a discutat despre setul de caractere ASCII şi setul de caractere de control. Acestea sunt introduse pentru manipularea unei informații care poate fi doar afişată pe ecran sau la imprimantă. Specific acestei informații este că de obicei manipulează în pachete de mai mult de un caracter. În aceste condiții trebuie folosit un tablou de caractere. Acesta este denumit şir de caractere. El are în plus față de un vector adăugat întotdeauna la sfârşitul datelor utile un caracter special '\0' numit terminator de şir. A fost introdus pentru a uşura scrierea funcțiilor (de bibliotecă sau utilizator) folosite în prelucrarea unui şir de caractere.

Şirurile de caractere au exact aceleaşi caracteristici ca oricare tablou monodimensional.

Inițializarea uni şir de caractere se poate face prin mai multe metode:

- la declarare, fără definirea dimensiunii tabloului

  char s[]=".doc"

  în acest caz dimensiunea tabloului va fi exact numărul de caractere cu care e inițializat şirul.
- la declarare cu definirea dimensiunii tabloului

  char s[20]="doi"

- direct în program : *s="Dati valoarea"*
- prin citire de la tastatură

  cu procedură: gets(s)
  cu funcție: scanf("%s",s);
- prin inițializare caracter cu caracter în cazul unei prelucrări în program

  for(i=0;i<strlen(s1);i++)s2[i]=s1[i]+0x25;

În ultimul caz apare o problemă, în cazul şirului s2 nu este pus automat caracterul '\0' şi deci trebuie OBLIGATORIU introdus explicit. În exemplul de mai sus acest lucru se realizează prin adăugarea instrucțiunii:

s[i]='\0';

Prezentăm mai jos un program simplu ce inițializează în cod, element cu element un şir de caractere apoi afişează diverse poțiuni din el:

```
#include <stdio.h>

void main(void)
        {
        char nume[5];       /* definesc un şir de caractere */

          nume[0] = 'M';
          nume[1] = 'i';
          nume[2] = 'h';
          nume[3] = 'a';
```

```
  nume[4] = 0;     /* caracterul ce marchează sfârşitul */

  printf("Numele este  %s\n",nume);
  printf("O litera din el este %c\n",nume[2]);
  printf("O parte din nume este %s\n",&nume[1]);
getch();
}
```

Introducere în programare

8.2. Tablouri bidimensionale. Implementarea noțiunii de matrice

Pentru calcule matematice tablourile bidimensionale implementează exact conceptul de matrice şi pot fi manipulate în cod aproape la fel ca în matematică.

Apare aceeaşi problemă relativ la modul în care alegem dimensiunea maximă a unei matrici ca şi în cazul vectorilor. Există cele două cazuri:

- Utilizatorul ştie exact cât este dimensiunea maximă a matricilor care vor fi folosite, acesta este cazul ideal care este relativ rar.
- Utilizatorul are nişte cerințe vagi şi programatorul prin discuție directă poate afla acest maxim sau poate să-şi facă o idee despre el caz în care va lua o valoare acoperitoare.

În orice caz oricare ar fi dimensiunea unui tablou el are un număr total de elemente care va fi alocat static. Acest număr se poate afla conform formulelor de mai jos:

- tablou monodimensional: Nr_elem=dim_vect*sizeof(tip_data);
- tablou bidimensional: Nr_elem=dim_x*dim_y*sizeof(tip_data);
- tablou tridimensional Nr_elem = dim_x* dim_y *dim_z * sizeof(tip_data).

Deja apare o problemă relativ la spațiul maxim care poate fi ocupat de totalul datelor definite static (adică cum am vorbit până acum), şi anume faptul că chiar sub modelul de compilare HUGE nu se poate aloca mai mult de 1Mo=1024*1024 octeți pentru date. Deci implicit apare automat o limitare asupra dimensiunii maxime care o poate lua un tablou.

Deci indiferent de cerințele utilizatorului nu se poate depăşi această dimensiune. În cazul în care specificațiile depăşesc aceste dimensiuni există două metode:

- metoda swap-ului;
- metoda listelor de ordin n, unde n este conform necesităților.

În general aceste metode sunt mixate chiar şi la nivelul aplicațiilor actuale. Metoda swap-ului se concretizează prin scrierea într-un fişier a datelor ce depăşesc aceste dimensiuni maxime. Practic dispozitivul de stocare devine un fel de prelungire a memoriei operative, având însă dezavantajul că timpii de acces sunt cam de 10 ori mai mari decât în cazul lucrului cu RAM, ceea ce va duce la scăderea vitezei de execuție a respectivei aplicații. Metoda listelor se bazează pe alocarea dinamică care ne permite accesul la toată memoria fizică RAM neocupată de alte programe.

Mai jos vom prezenta un scurt exemplu de folosire a tablouri pentru implementarea de operațiuni simple cu matrici.

```
#include <stdio.h>
#include <conio.h>

void main(void)
    {
    int m1[5][5],m2[5][5],m3[5][5],m4[5][5],i,j,k,m,n,p;
    clrscr();
    printf("n="); scanf("%d",&n);// se poate scrie şi aşa în cod
    printf("m="); scanf("%d",&m);// dar nu este recomandat
    printf("p="); scanf("%d",&p);// am citit dimensiunile matricilor
    for(i=0;i<n;i++)
        for(j=0;j++;j<m)
        {
        printf("\nA[%d,%d]=",i,j);
        scanf("%d",&m1[i][j]);
        }// am terminat de citit prima matrice
    // se observă ca numele sub care o citesc nu are legătură cu
    //modul în care o denumesc în program.
    for(i=0;i<m;i++)
        for(j=0;j++;j<p)
        {printf("\nB[%d,%d]=",i,j);
        scanf("%d",&m2[i][j]);        }// se poate scrie şi aşa în cod
    // deci am citit cele două matrici, acum vom începe procesarea.
    for(i=0;i<n;i++)
        for(j=0;j++;j<p)
        {
        m3[i][j]=0;
        for(k=0;k++;k<m)
                m3[i][j]+=m1[i][k]*m2[k][j];
        }// înmulțirea a două matrici
    printf("Matricea Cnp=Amn*Bnp este:")
    for(i=0;i<n;i++)
        {printf("\n")
        for(j=0;j++;j<p)printf("%d ",m3[i][j]);
        }
    for(i=0;i<n;i++)
        for(j=0;j++;j<p)
        m4[i][j]=m3[j][i];
    printf("Transpusa lui C este ");
    for(i=0;i<p;i++)
        {printf("\n")
        for(j=0;j++;j<n)printf("%d ",m4[i][j]);
        }
    getch();
    }
```

Deoarece tablourile bidimensionale de caractere nu sunt folosite uzual, nu le vom trata.

9. Tipuri de date definite de utilizator

S-a discutat până acum, despre tablouri de date mono sau multidimensionale care însă conțineau numai date de acelaşi tip. Sunt multe cazuri în care este nevoie ca date de tipuri diferite să fie grupate sub un nume comun.

Această problemă s-a rezolvat prin introducerea noțiunii de structură de date, care permite utilizatorului să-şi creeze noi tipuri de date cât mai apropiate de obiectele din lumea reală. Acest lucru se mai numeşte abstractizarea datelor.

La fel cum funcțiile permit descompunerea complexității unui program în termeni de operațiuni pe date, structurile sunt folosite pentru abstracțiuni.

9.1. Declararea unei structuri

O structură este un tip de dată generalizat definit de utilizator, care poate fi folosit pentru gruparea diferitelor tipuri de date într-un nou concept. De exemplu un tip de date ce se referă la unele din caracteristicile unui student este prezentat mai jos.

```
struct Student
    {
    char name[20]; // numele studentului
    unsigned short int age; // virsta
    unsigned char n_lab; // nota la laborator
    unsigned char n_ex; // nota la examen
    }
```

Există o convenție comună legată de numele unei structuri şi anume acesta să înceapă cu literă mare. Un exemplu de declarații cu acest tip de date este dat mai jos.

```
    struct Student tmp, an99[20];
```

9.2. Operatorul typedef

Comanda typedef care este folosită la redenumirea unor variabile enumerate, poate fi folosită şi împreună cu cuvântul cheie struct. Pentru o structură trebuie să apară, după ce este definită, şi înainte de orice declarație de variabile. Uzual noul tip de date se redenumeşte (i se ataşează o etichetă echivalentă) ca mai jos:

```
typedef struct Student
{
char name[20]; // numele studentului
unsigned short int age; // virsta
unsigned char n_lab; // nota la laborator
unsigned char n_ex; // nota la examen
} stud;
```

Deci noul tip de date a fost redenumit ca stud, în aceste condiții declarații relative la el devin:
Un alt exemplu de definire Ş
Stud tmp,an99[20];

- Definesc noul tip de date

```
struct my_data
{
char name[19];
secol[11];
short month,day,year;
}
```

- Folosesc noua definiție

```
struct my_data test,tab[100],*ptab;
```

Deoarece my_data nu este un tip predefinit de date ,ci este un conglomerat definit de programator trebuie specificat şi cuvântul cheie struct.

Creearea unui nou tip de date se face astfel:

```
typedef  struct
        {
        char name[19];
        secol[11];
        short month,day,year;
        }pers;
```

Începând de acum pers se va folosi direct fără alte specificații în definirea unor elemente de acest tip.

```
pers test,tab[100],*ptab;
```

Inițializarea unei structuriS
Ca şi în cazul tablourilor sau altor tipuri de date, poate fi realizată astfel:
→ La declararea unui element
Ex: pers eu={“MIKE”, “354263”,3,5,1978};
→ Oriunde in cadrul programului
Obs: Inițializarea unei structuri nu poate fi realizată la definirea noului tip de dată (cu typedef).Evident, există o diferență clară intre definirea unui nou tip de dată şi definirea unei variabile de un anumit tip.

9.3. Accesarea membrilor unei structuri

Deoarece acest tip de date este creat de programator compilatorul sau funcțiile existente nu ştiu să lucreze decât cu tipurile predefinite de date. De aceea este necesar ca

prelucrarea să se facă tot la nivel de membri (câmpuri). Pentru aceasta trebuiesc construite funcții specializate în operațiunile cu noul tip de date.

În cazul declarației unui element de tipul respectiv operatorul de acces la oricare din membri este '.'. Accesul se face prin referire la numele global urmat de numele membrului ca în exemplul de mai jos unde este realizată citirea de la tastatură a unui tablou de elemente de tip student.

```
…
for(i=0;i<10;i++)
    {
    clrscr();
    printf("Nume");
    scanf("%s", an99[i].nume);
    printf("\n Nota la laborator:");
    scanf("%d",an99[i].n_lab);
    printf("\n Nota la examen:");
    scanf("%d",an99[i].n_ex);     }
```

9.4. O structură în interiorul altei structuri (nested structures)

Există cazuri când este necesar ca informația să fie cât mai ierarhizată obținându-se astfel un cod cât mai clar. În cazul din exemplului anterior, dacă doresc să realizez o aplicație pentru un secretariat, am nevoie şi de numărul grupei. Există două metode de a rezolva problema. Una ar fi introducerea a încă unui câmp care să rețină numărul grupei, dar mod de rezolvarea poate duce pe măsură ce tot mai adaug caracteristici noi în structură la un cod greu de urmărit. Altă cale ar fi crearea unei noi structuri după cum este prezentat mai jos.

```
typedef struct Grupa
        {
        int nume; // numarul grupei
        int an; // an de studii
        stud t[35]; grupa contine maximum 35 de studenti
        }gr;
gr temp;
```

Accesul la membri în acest caz se face prin compunere ca mai jos

```
temp.nume //umele grupei
temp.t[3].name      // numele celui de-al treilea student din grupă
```

Mai jos prezentăm încă un exemplu

```
typedef  struct
         {char name[19],snum[11];
         struct
                 {short day;
                 short month;
```

```
        short year;
        }birth_date
    }pers;
```

O referire la day se va face pentru persoana eu: eu.birth_date.day

Element structură mare → substructură → câmp

Obs: numele câmpurilor unei structuri nu sunt văzute direct în afară şi deci o referire:

struct x {int x;) x; este legală la accesare x.x=5.

Atenție ! Începătorii să evite acest lucru deoarece crează o ambiguitate nerecomandată în cod.

9.5 Structuri autoreferite

O structură NU poate conține instanțieri ale ale ei, dar poate conține pointeri către instanțieri ale ei.

```
struct p
    {int a,b;
        float c;
        struct p next;
    }
```

la analiza lexicală se intră în ciclu infinit

```
struct p
    {int a,b;
        float c;
        struct p* next;
    }
```

OK!!Acest lucru este posibil deoarece pointerul este o variabilă care reține adresa altei variabile

9.6 Transferul structurilor ca argumente de funțiiS

→ transfer chiar structura (prin valoare)

→ transfer un pointer către ea (prin referință)

Ex: pers om;
- func1 (om)
- func2 (&om)

Transferul prin valoare se recomandă in următoarele cazuri:

- structura este mică
- garantarea faptului că funția apelantă nu modifică structura apelată

La apelul prin valoare compilatorul generează o copie a respectivei structuri (funcția apelantă poate deci modifica doar această copie).

Dacă am dori să specificăm transferul prin referință in clar (vezi func2(&om)) este suficient ca să declarăm pers* om şi atunci func2(om) este corect.

Atunci în acest caz func1(om) nu ami este corect pentru că om este un pointer şi nu o valoare.

9.7 Returnarea unei structuri

K&R nu permite returnarea unei structuri dar ANSI permite acest lucru.

Deci:

pos by value

```
pers f(….)
{
        pers x;
        …
        return x;
}
```

pos by reference

```
pers* f1(…..)
{
        static pers* p;
        …
        return &p;
}
```

9.8. Câmpuri de biţiS

Membrii unei structuri de tip int sau unsigned pot fi alocaţi în câmpuri de 1 până la 16 biţi. Biţii sunt alocaţi începând cu cel mai puţin semnificativ bit al fiecărui cuvânt de 16 biţi. Sintaxa este:

tip [identificator] ":" lărgime ";"

Lărgimea câmpului trebuie să fie o expresie întreagă constantă cuprinsă intre 0 si 16 (0 înseamnă aliniere la următoarea limită de cuvânt). Dacă nu se specifică un nume pentru câmpul de biţi, câmpul este alocat dar nu este accesibil (se foloseşte pentru alinieri).

```
struct special_word
    {
    int      i :2 ; // va fi reprezentat pe biţii b0,b10
    unsigned  j :5 ; // va fi reprezentat pe biţii b2 până la b60
    int        :4 ; // va fi reprezentat pe biţii b7 până la b10S
    int      k :1 ; // va fi reprezentat pe bitul b11
    unsigned  m :4 ; // va fi reprezentat pe biţii b12 până la b15
    } ;
```

O restricţie importantă asupra operaţiilor care se pot face cu câmpurile de biţi este aceea că nu li se poate afla adresa cu ajutorul operatorului & (adresa unui bit nu are sens).

9.9. Uniuni

Uniunile corespund cu înregistrările cu variante din alte limbaje de programare (Pascal, Modula-2, etc.); o uniune este un tip "hibrid", valoarea variabilei uniune fiind valoarea membrului activ al uniunii, ceilalţi membri fiind nedefiniţi. Din punct de vedere sintactic definiţia unui tip uniune seamănă cu definiţia unui tip structură cu diferenţa că indicatorul struct se înlocuieşte cu union (sunt permise şi câmpuri de biţi).
Ca implementare sunt similare cu structurile, cu diferenţa că membrii sunt suprapuşi unul după altul în memorie. Spaţiul de memorie este folosit la maximum în raport cu

structurile. De fapt, permit ca aceeaşi zonă de memorie să fie interpretată diferit fără a folosi **cast**.

Dimensiunea spațiului de memorie alocat uniunii este egală cu dimensiunea spațiului alocat celui mai mare membru. Accesul la membrii uniunii se face sub forma unor componente selectate, folosind aceiaşi operatori şi aceleaşi reguli ca în cazul structurilor. Numele de uniuni respectă aceleaşi reguli ca şi numele de uniuni (spațiu de nume, utilizare etc.).

Înregistrări variabile

În cazurile în care se poate ca într-o bază de date, două sau mai multe câmpuri să se autoelimine reciproc (funcție de datele introduse în ea) se poate obține optimizarea spațiului grupat în memorie de un element al structurii prin gruparea acelor câmpuri într-o uniune ce aparține structurii.

Ex.:

```
        typedef union
        {
                struct
                {
                char c1,c2;
                }s
        long j;
        float x;
} u
```

1000	1001	1002	1003
C1	C2		
J			
X			

Compilatorul alocă suficientă memorie pentru a putea reține cel mai mare membru şi toți membrii încep la aceeaşi adresă. Datele stocate într-o uniune depind de care membru al uniunii se va utiliza.

pas.s.c1=’a’
pas.s.c2=’b’

1000	1001	1002	1003
a	b	...	Unused

pas.j=5 suprascrie cele 2 caractere şi depune 5 pe toți cei 4 octeți (tip long int)

1000	1001	1002	1003
0	0	0	5

Suportă aceleaşi reguli sintactice ca structurile.

Există 2 aplicații de bază pentru uniuni:

1. creează structuri flexibile (varianta records în care pot reține diferite tipuri de date)
2. interpretează aceeaşi memorie în moduri diferite

Exemplu pentru cazul 1:

```
Union myunion
    {
    int i ;
    double d ;
    char c ;
    } mu ;

sizeof ( myunion ) --> 8
sizeof ( mu ) --> 8
sizeof ( mu .i ) --> 2
sizeof ( mu .c ) --> 1
sizeof ( mu .d ) --> 8
```

Respectând conversiile necesare, putem spune că un pointer către o uniune este un pointer către oricare dintre membrii ei:

*(double *) & mu == & mu .d*

O uniune nu poate fi inițializată decât prin intermediul primului său membru declarat. Vom prezenta mai jos un program care citeşte cuvintul special care reține componentele hardware de bază instalate într-un sistem de calcul. Acesta este inițializat în urma autodetecției făcute în secvența de pornire ce are loc la punerea sub tensiune a sistemului.

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>

union
    {
    unsigned w;
    struct bit_field {
     unsigned fd :1;
     unsigned dimR :2;
     unsigned modV :2;
     unsigned nrF :2;
     unsigned DMA :1;
     unsigned nrRS :3;
     unsigned game :1;
     unsigned serP :1;
     unsigned nrLpt :2;
```

```
            } b;
        } eqp;

void main(void)
        {
        unsigned far *ptr;

         clrscr();
         ptr=MK_FP(0,0410);
         eqp.w=*ptr;
        // urmează afişarea fiecărui element al uniunii conform
        // cu înțelesul dat
        …
        getch();
        }
```

Cazul 2 Interpretarea diferită a datelor

Ca un exemplu să considerăm problema comunicației seriale, unde datele vin octet cu octet.

Uniunile permit o modalitate de grupare, a octeților, împreună astfel încât să poată fi reconstruite în forma lor originală. De exemplu, să presupunem că get_byte( ) este o funcție care întoarce un octet citit dintr-un device de comunicație.

O valoare double (pe 8 octeți) poate fi extrasă din device prin 8 apeluri recursive ale funcției după cum se vede:

```
union double
{
        char c[];
        double val;
}

double get_double( )
{
        extern char get_byte( );  //optimal – depinde de modul de structurare a
                                  //fişierului_______
        int j;
        union double d;
        for (j=0; j<8 ; j++)
                d.c[j]=get_byte( );
        return d.val;
}
```

Atenție!!! Este o diferență majoră între acest mod de interpretare şi conversiile prin **cast**.

Inițializarea uniunii (specifică ANSI):

```
union ex
        {int I;
        float f;
```

```
        };
    union ex test={1};
```

sau în cazul uniuni ce reține şi o structură

```
    union n
        {
        struct {int i; float f;}s;
        char ch[6];
        }
    union n test1 = {1, 1.10}
```

9.10. EnumerăriS

Tipurile de date enumerate sunt folosite în principal pentru a da nume mnemonice unor constante întregi. În exemplul de mai jos

enum zile { luni,marti,miercuri,joi,vineri,simbata,duminica } oricezi ;

se defineşte un nou tip întreg, numit enum zile, un număr de constante de acest tip (luni, marți, ...) cu valori de acest tip, şi se declară o variabilă de tip enum zile, numită oricezi.

În prezenta opțiunii -b (sau a comenzii corespunzătoare din mediul integrat) compilatorul are libertatea sa aloce 1 sau 2 octeți pentru o variabila de tip enumerare (in funcție de valorile constantelor definite); valorile constantelor definite sunt de tip unsigned char sau int. În C, unei variabile de tip enumerare i se poate atribui orice valoare întreagă;

```
enum days this_day ;        // legal si in C si in C++
days another_day ;          // legal numai in C++
```

Constantele de enumerare au implicit valori începând cu 0 care cresc din 1 in 1 (luni = 0, marți = 1, ..., duminica = 6). La definiția tipului, orice constantă de enumerare poate fi eventual urmată de un inițializator, sub forma unei expresii întregi constante (pozitive sau negative), care stabileşte valoarea constantei. Constantele de enumerare următoare care nu au o valoare explicită cresc din 1 in 1. Sunt permise valori duplicate. Numele constantelor de enumerare se află în acelaşi spațiu de nume ca şi numele de variabile, funcții şi tipuri. Valori de tip enumerare pot apare oriunde sunt permise valori de tip întreg.

```
enum coins
    {
    penny = 1,                  // 1
    tuppence,                   // 2
    nickel = penny + 4,         // 5
    dime = 10,                  // 10
    quarter = nickel * nickel   // 25
    } ;
```

Tipuri enumerative

Se folosesc atunci cand se doreste creerea unui set mic de valori care pot fi asociate cu o variabila .Compilatorul va genera o eroare daca se incearca asigurarea unei valori care nu face parte din multimea valorilor specificate in declaratii.

De ex.:

enum { red,blue,green,yellow } my_colors
enum { bright,medium,dark } my_intensity

color = yellow *OK*
color = bright *ERROR*

Aceste valori numerice pot fi si direct specificat ,caz in care compilatorul aplica sumarea cu 1 poziție numai la cele nespecificate.
De ex.:

enum { apple,arong=10,lemon,grope=-5,melon }

va avea ca rezultat urmatoare atribuire de catre compilator:

apple=0;
orange=10;
lemon=11;
grape=-5;
melon=-4;

CAPITOLUL 10

10. Noțiunea de funcție

Definiție: O funcție C este o colecție de operații specifice limbajuluiS

Până acum s-a discutat despre programe extrem de simple. De asemenea s-au folosit apeluri de funcții predefinite, al cărui cod este gata scris. Se poate spune că o funcție este o bucată de program separată (subprogram) care este folosită în mai multe scopuri:

1. Realizarea unui cod sursă cât mai clar.
2. Simplificarea proiectării unei aplicații (prin structurare).
3. Realizarea de biblioteci de funcții, deci creşterea vitezei de dezvoltarea a aplicațiilor ulterioare

Pentru introducerea acestui concept în mod corect trebuie discutate întâi un minim de tehnici de proiectarea a unei aplicații şi aceasta deoarece o funcție nu se introduce în cadrul unui program din motive "artistice" ci are o justificare la nivel logic foarte clară.

Ca tehnici de proiectare există două metode: *top-down* (de sus în jos) şi *bottom-up* (de jos în sus). Proiectarea unei aplicații constă în descompunerea logică pe submodule.

Funcție de complexitatea acestei aplicații descompunerea poate continua recursiv la nivelul fiecărui modul rezultat până când:

- acesta este extrem de simplu de realizat
- este deja implementată respectiva parte într-o bibliotecă a compilatorului
- este deja dezvoltată anterior
- datorită simplității programului nu mai are sens.

În cazul metodei de proiectare top-down proiectarea decurge exact cum am descris mai sus. Uzual în programarea C se foloseşte această metodă. În cazul bottom-up se porneşte de la un set de funcții şi operațiuni existente şi se construiesc module care la rândul lor vor construi recursiv alte module, până când rezultă aplicația. Această metodă este folosită uzual în cazul proiectării aplicațiilor pentru bazele de date.

Totuşi este greu de impus un anumit tip de gândire când se realizează proiectarea unei aplicației. Există o teorie foarte bine fundamentată şi documentată privind aceste probleme. Într-o primă fază esențiale sunt experiența avută în proiectarea unui anumit tip de aplicație, şi apoi experiența de programare.

Deja pentru aplicații foarte mari se lucrează în echipe cu membri extrem de specializați în câte o fază a procesului de proiectare, implementare, testare. Însă indiferent cât de complicată ar fi distribuția unei echipe programatorului propriu zis i se dau fie specificațiile relativ la ce va trebui să facă partea de cod care o dezvoltă fie în anumite cazuri algoritmul. În ambele situații modul în care îşi descompune aplicația pentru a o putea realiza nu priveşte pe nimeni. La sfârşit aceasta trebuie să funcționeze corect şi atât.

Pasul imediat următor în aplicarea metodei top-down este rafinarea paşilor

Exemplu: Cum s-ar proiecta un program care să "Numere cuvintele dintr-un fişier" ?

Descriere în pseudocod

```
Deschide fişierul
atâta timp cât mai sunt cuvinte in fişier fă
{
        citeşte un cuvânt
        creşte numărul de cuvinte cu 1;
}
Închide fişierul.
Afişează numărul de cuvinte
```

Rafinarea paşilorS

```
Deschide fişierul
Atâta timp cât mai sunt cuvinte in fişier
{
        citeşte caractere până se întâlneşte un caracter diferit de ' '
        citeşte caracterele până găsesc un ' '
        incrementează( nr_cuv )
}
Închide fişierul
Afişează ( nr_cuv )
```

10.1. Declararea şi definirea funcţiilorS

Funcţia *main()* a fost folosită în toate exemplele de până acum. De câte ori un program C este executat el începe cu prima instrucţiune din funcţia *main()*. Bibliotecile C ne pun la dispoziţie o sumă variabilă de rutine precompilate. Totuşi pentru a putea rezolva probleme mai complexe este necesar să se proiecteze (declarare şi definire) propriile funcţii.

10.1.1. Declararea funcţiilorS

O funcţie este declarată exact ca o variabilă, iar tipul de dată pe care-l returnează trebuie specificat. În C trebuie avut grijă ca numele unei funcţii proprii să nu coincidă cu numele unei funcţii de bibliotecă. În caz contrar link-editorul nu va şti la generarea de legături între fişierele obiect, dacă apelul respectivei funcţii se referă la rutina precompilată existentă deja sau la corpul funcţiei proprii. Se poate chiar mai rău să nu se genereze un mesaj de eroare, acest lucru se întâmplă când prototipurile celor două funcţii se potrivesc, iar acest tip de eroare este extrem de greu de detectat.

De asemenea trebuie specificate şi argumentele ce vor fi transmise funcţiei la apel (tipul, ordinea lor). O formă generală pentru declararea unei funcţii este:

```
<tip_dată> nume_funcţie(<listă_argumente>);
```

unde :

<listă_argumente>={tip_dată nume_arg1, tip_dată nume_arg2, … tip_dată nume_argn}.

Funcțiile sunt declarate înainte de funcția *main()*. Acest lucru permite ca oricare funcție să fie apelată de oricare alta inclusiv de ea însăşi (cazul funcțiilor recursive).

Se pot declara funcții şi în interiorul altor funcții dar în acest caz ele nu pot fi apelate decât în interiorul funcției unde au fost declarate. De aceea metoda de declarare globală a lor este cel mai des folosită.

Este necesară să se realizeze specificarea tipului de dată pentru argumente şi valoarea returnată. Acest lucru permite compilatorului să verifice dacă funcțiile sunt apelate corect şi să realizeze orice conversie implicită ar fi necesar.

De exemplu un apel de funcție ca mai jos:

```
#include <math.h>
...
double x;
...
x=sqrt(4);
...
```

poate fi considerat ca incorect deoarece funcția radical are ca parametru de intrare o variabilă de tip double. Datorită faptului că funcția este declarată în cadrul bibliotecii math.h care este inclusă în program, compilatorul va face conversia de la întreg la double a valorii 4 astfel încât la apel funcția radical va primi o valoare corectă (4.0).

10.1.2. Declarații implicite

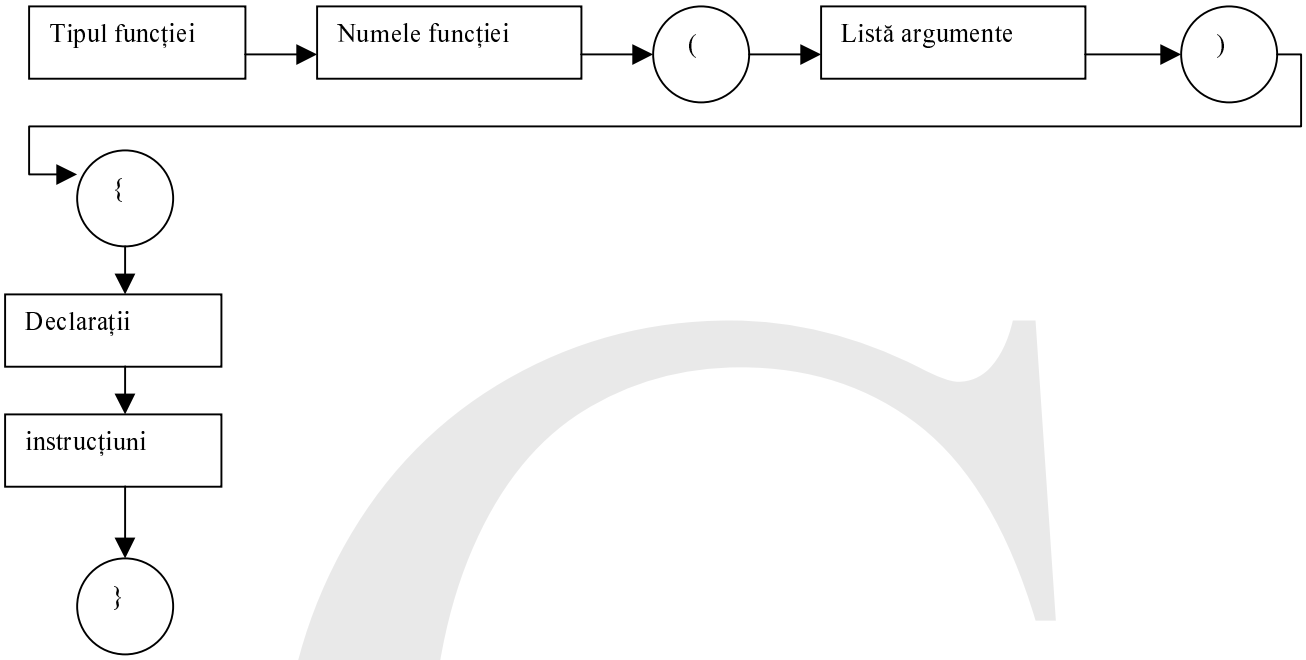
Dacă nu este specificat nici un tip de dată la valoarea pe care trebuie să o returneze o funcție atunci compilatorul va considera că funcția returnează un întreg. Similar dacă funcția nu este declarată, compilatorul va considera că returnează un întreg şi va da eroare dacă în program este folosită altfel.

Totuşi dacă se ține cont numai pe tipul implicit compilatorul nu va şti să facă conversiile de tip la apel.

10.2. Definirea funcțieiS

Definirea unei funcții se referă la o bucată de cod C care rezolvă o anumită problemă (aceea pentru care s-a introdus funcția). Această definiție se face în afara corpului funcției main() sau al altei funcții şi are forma generală dată mai jos:

Structura generică a unei funcții C



Exemplu:

```
   tip      nume      declarație argument
int        pw2        ( int num )
                            nume argument
{
 int rez;          declarație variabilă locală
rez = num * num;
return rez;          instrucțiuni C
}
```

corpul funcției

sau:

<tip_dată> nume_funcție(<listă argumente>)
{
<declarații variabile locale>
<instrucțiuni>
return <variabilă/expresie>
}

Prima linie trebuie să fie identică cu cea din declarație. Grupul de variabile locale (automate) care pot fi folosite în interiorul unei funcții constă în cele declarate în lista de argumente sau la începutul funcției.

Ca şi în cazul funcției *main()* variabilele trebuiesc declarate imediat după paranteza deschisă şi înainte de alte instrucțiuni.

Ca un exemplu de funcție să luăm cazul calculului rădăcinii pătrate dar cu specificarea preciziei dorite. Se va folosi metoda Newton Raphson pentru aproximarea rădăcinii.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>

double radical(double x,float precizie);

void main(void)
    {
    float nr, prec;
    double rezultat;
    clrscr();
    printf("Dati numarul");
    scanf("%f",&nr);
    printf("\nDati precizia ");
    scanf("%f",&prec);

    switch((rezultat=radical(nr,prec)))
        {
        case –1.0: printf("Eroare, Numar Negativ"); break;
        case –2.0:printf("Eroare, Precizia este intre 0 si 1 ");break;
        default: printf("\nRadicalul este %lf",rezultat);
        }
    printf("\n Press any key to continue!");
    getch();
    }

double radical(double x,float precizie)
    {
    double x_vechi, x_initial=x;
    if(x<0.0)return –1;
    if(!x)return x;
    if((precizie>1)||(precizie<0))return –2;
    do
    {
        x_vechi=x;
        x=0.5*(x+x_initial/x);
    }while(fabs((x-x_vechi)/x)>precizie);
    return x;
    }
```

Se observă că aproape jumătate din corpul funcției tratează erorile ce pot apărea. Deşi la prima vedere acest lucru apare ca un efort suplimentar, nejustificat în programare, în momentul în care este cazul unei aplicații cu peste 20 funcții care se apelează unele, pe altele codul este relativ greu de urmărit şi depanat în caz contrar.

În condițiile în care s-a prevăzut fiecare funcție de la început cu aceste verificări atunci este extrem de uşor ca să se poată depista erorile. Dacă această politică este aplicată la orice nivel al programului va conduce la scrierea unui cod stabil. Stabilitatea se referă la faptul că dacă apare undeva în cursul execuție o eroare datorată fie condițiilor externe (intrări greşite) fie condițiilor interne (de exemplu eroare de programare sau virus) aceasta va fi semnalată utilizatorului. Ea se va propaga pe cale ierarhică din apel în apel de funcție, fără ca programul să iasă din parametri normali de funcționare, nu se pierde controlul asupra aplicației.

Pentru realizarea unor programe care să primească parametri de intrare din linia de comandă funcția *main()* poate primi un set de parametri care să conțină exact sirurile de caractere din linia de comandă. Mai jos este dat un exemplu simplu de folosirea a acestei facilități şi anume un program care va afişa parametrii primiți prin linia de comandă.

```
#include <stdio.h>
void main(int argc, char **argv)
{
    int i;
    printf("argc = %d\n\n",argc);
    for (i=0;i<argc;++i)
     printf("argv[%d]: %s\n",i, argv[i]);
}
```

Se observă că primul *arc* specifică numărul de lemente din care este compusă linia de comandă, iar *argv* reține adresa de început a zonei de memorie în care sunt stocate acestea.

10.3. Tipul de dată void

Geneza pointerului void

A fost introdus de standardul ANSI pentru a rezolva următoarea ambiguitate de sintaxă:

Ştim că avem posibilitatea de a aloca memorie.

Funcția întoarce un pointer către începutul zonei de memorie alocată (indiferent că suntem sub DOS, WINDOWS sau UNIX). Întrebarea este de ce tip este acest pointer sau, tradus, cum va fi interpretată zona contigua de memorie rezervată .

1) Ca fiind grupată în elemente de lungime 1 octet(signed sau unsigned sau char)
2) Ca fiind grupată în elemente de lungime 2 octeți (signed sau unsigned)
   ş.a.m.d.

Pentru a rezolva această semiambiguitate d.p.d.v. al compilatorului şi nu numai, s-a decis ca pointerul returnat să fie de tip void. Un tip inexistent, dar convertibil fie explicit prin cast (K&R standard), fie implicit prin asignare (ANSI) la tipul variabilei careia i se asignează.

În limbajul C o funcție trebuie să returneze un tip de date la terminarea execuției. Totuşi există cazuri când nu se fac calcule directe ci se realizează anumite acțiuni cum ar

fi de exemplu afişarea unui mesaj. În aceste cazuri, totuşi trebuie specificat compilatorului faptul că funcția nu returnează nimic.

Există şi cazul invers adică o funcție care nu primeşte nici un parametru dar returnează rezultate (cazul citirii unui port de date). Pentru a putea specifica aceste situații se foloseşte tipul de dată void care în traducere înseamnă: lipsit de conținut, nimic.

Transferul Argumentelor

Argumentele unei funcții, reprezintă căile de transfer a datelor către funcție. Multe limbaje de programare transferă argumentele prin referințe, ceea ce înseamnă că trimit un pointer către argument. În C argumentele sunt trimise prin valoare ceea ce înseamnă că o copie a argumentului este trimisă funcției. Funcția poate modifica valoarea copiei, dar nu poate modifica valoarea care o are argumentul în rutina apelantă.

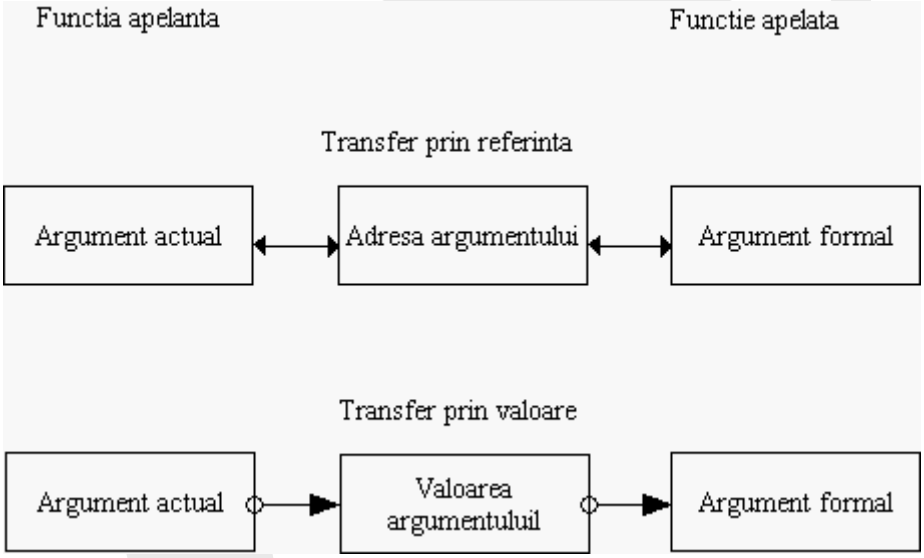


Figura arată diferența; se observă că săgețile de la transferul prin referință, sunt bidirecționale, ceea ce specifică faptul că modificările locale se repercutează direct.

Argumentul trimis este de obicei numit argument actual, în timp ce copia recepționată este numită argument formal sau parametru formal.

Deoarece C transferă argumentele prin valoare, o funcție poate asigura noi valori la argumentele formale fără a le afecta pe cele actuale. De exemplu:

```
#include<stdio.h>
void main(void)
        {extern void f( );
        int a=2;
        f(a);
        printf("y=%d \n ", a);
        exit(0);
        }
```

```
void f (int arg)
        {
        arg=3;
        }
```

va avea ca rezultat afişarea valorii 2, modificarea copiei nemaiafectând valoarea din funcția apelantă. Dacă se doreşte ca funcția să poată schimba valoarea unui obiect, atunci trebuie ca aceasta să primească adresa obiectului, deci să i se trimită un pointer la obiect, ca în exemplul următor:

```
void swap(int * x,int* y)
        {
        register int temp;
        temp=*x;
        *x=*y;
        *y=temp;
        }
void main(void)
        {
        int a=2,b=3;
        swap (&a, &b);
        printf("a= %d , b=%d\n",a,b);
        }
```

unde se va afişa a=3 şi b=2;

Cu transferul prin referință ne-am întâlnit de la primele lecții: scanf("%d",&nr).

Declarații şi apeluriS

Funcțiile pot apărea într-un program

→ Definiție: o declarație care defineşte ce face funcția precum şi numărul şi tipul argumentelor ei.

→ Function allusive: se declară o funcție definită în altă parte. Acest lucru specifică ce tipuri de valori întoarce funcția, precum şi ce tipuri de valori primeşte

→ Apelul funcției : apel propriu-zis la întâlnirea lui în contextul execuției programului Sare la zona corpului funcției execută ce este acolo şi apoi se reîntoarce la locul apelului.

Declararea argumentelor

Declararea argumentelor se supune la aceleaşi reguli ca oricare declarare de variabile cu următoarele excepții:

1. singura clasă de stocare permisă este register
2. char şi short sunt convertite la întregi; float sunt convertite la double. Noile facilități ANSI C permit deyactivarea acestor conversii aotomat
3. un argument formal declarat ca funcție este convertit la pointer către funcție
4. se poate să nu includ un inițializator într-o declarație de argumente

Corpul funcției

Corpul unei funcții este delimitat de un set de paranteze {}. El poate fi gol, fapt ce este folositor în diverse perioade a dezvoltării unui program.

Unul din primele lucruri în cazul proiectării unei aplicații mari, este definirea unui set de operații de nivel înalt ce corespund ca funcționare. În aceste stagii este folositor să am un program, funcția, declarații, parametrii specificați, dar corpul gol. La apel ea nu va executa nimic. Ulterior aceste corpuri vor fi completate.

Valorile returnate (întoarse)

O funcție poate returna o singură valoare prin intermediul cuvântului cheie return. Această valoare poate fi un pointer pentru tablou(valori multiple) sau o funcție. Indirect însă şi acestea pot fi întoarse, dacă se foloseşte un pointer la acestea ca valoare atinsă. O structură sau uniune pot fi returnate direct, deşi acest lucru nu este recomandat datorită dimensiunilor mari ale lor.

Corpul unei funcții poate conține oricâte apeluri de return. La primul întâlnit, execuția funcției este încheiată; în lipsa totală a unui return, întorcerea în funcția apelantă, se face atunci când se întâlneşte marcatorul de sfârşit de bloc " } ".

Valoarea întoarsă trebuie să fie compatibilă cu tipul funcției. Ex. de caz în care valoarea întorsă este diferită, este dat mai jos, dar în acest caz conversia la valoarea la valoarea returnată se face automat.

ATENȚIE!!! Acest lucru poate crea o groază de neplăceri fiind o eroare extrem de greu de găsit.

```
Float f(void)
{
        int a;
        char c;
        float f2;
if ( a > (c+0x30))
        {
        f2=a;   // a convertit la float
        return a;
        }
else
        {
        f2=c;  //c convertit la float
        return c;
        }
}
```

Limbajul C este foarte riguros relativ la potrivirea tipurilor de pointeri. În exemplul următor f( ) este declarată ca o funcție ce întoarce un pointer către char, sunt prezente unele posibilități legale sau ilegale de întoarcere de rezultat.

```
Char* f( )
{
        char **cm, *cv, *cv1, ca[10];
        int *ip, *ip1;
        cv=cv1;    //admis, au acelaşi tip
        return cv1;  //admis, cv1 este char*
```

sau

```
        cv1= *cm;   // admis, au acelaşi tip
        return *cm;  //admis, *cm este char*
```

sau

```
        cv=ca;   //admis, au acelaşi tip
        return ca; //admis, ca este char*
```

sau

```
        cv=*cv1; //eroare, cv este char*, iar *cv1 este char**
        return *cv1; //eroare din acest motiv
```

sau

```
        cv1=ip1; //eroare char* ≠ int*
        return ;   // Atenție!!! Produce un comportament impredictibil, se aşteaptă  o
                   // valoare să fie întoarsă
```

Aluzii la funcțiiS

O aluzie la funcție reprezintă o declarație a unei funcții care este definită în altă parte, de obicei într-un fişier sursă diferit. Implicit toate funcțiile sunt presupuse a returna int.

Scopul principal al redeclarării este de a indica prototipul (ca compilatorul să ştie cum să transmită variabilele la apelul ei), precum şi clasa de stocare. În cazul în care clasa de stocare nu este precizată, ea este considerată extern implicit. Acest lucru înseamnă că definiția funcției precum şi corpul ei pot apărea în oricare din fişierele sursă a aplicației, inclusiv în cel curent. Dacă se omite şi specificarea tipului, atunci implicit este int.

Scopul în cazul unei aluzii la funcție este tipic ca la oricare altă variabilă, apelurile în bloc au scop bloc, restul au scopul fişier.

Conversiile automate de argument

În absența prototipului (alt motiv pentru aluziile la funcții) toate argumentele mai mici ca int sunt convertite la int, iar toate argumentele float sunt convertite la double. Deci poate fi o sursă bine ascunsă de erori.

Pointeri la funcțiiS

Sunt instrumente puternice pentru ca ne permit o modalitate elegantă de apel pentru diferite funcții, bazându-se pe datele de intrare.

Înainte de a discuta pointerii la funcție, trebuie vorbit puțin despre modul în care compilatorul interpretează declarațiile funcțiilor precum şi apelurile acestora.

Sintaxa pentru declararea şi invocarea funcțiilor este foarte apropiată de sintaxa folosită în cazul tablourilor. De ex. în declarația int v[5], v este pointer către primul

element al tabloului. Atunci când simbolul este urmat de un nume/număr între paranteze, pointerul este indexat cu acel număr apoi referit. Un proces similar apare şi în cazul funcțiilor.

Fie declarația int f( ); în acest caz f este un pointer la funcție. Când el este urmat de o listă de argumente inclusă între paranteze, el este referit (alt mod de a spune că o funcție este apelată). De asemenea, dacă int v[5] se referă la un pointer constant la fel şi în cazul int f( ), f este tot un pointer constant, însă este ilegal să se asigneze o valoare lui f..

Pentru a declara o variabilă pointer către o funcție, numele pointerului trebuie precedat de *; de exemplu int (*pf)( ) este un pointer către o funcție ce întoarce un întreg. De fapt, se declară o variabilă de tip pointer, care este capabilă să rețină un pointer către o funcție ce întoarce un întreg.

Parantezele sunt necesare, deoarece în lipsa lor pf ar fi considerată o funcție care întoarce un pointer la întreg.

Asignarea unei valori pentru pointerii la funcții se realizează astfel: fie funcția int fat_boy( ) şi pointerul la o funcție ce întoarce un întreg int (*pf) ( ). Asignarea se relizează direct pf=fat_boy. Devin astfel clare următoarele erori:

1. pf=f1( ) unde pf este pointer la funcție iar f1( ) este un întreg
2. pf=&f1( ) nu se poate obține adresa rezultatului unei funcții deoarece transferul se face prin stivă
3. pf = &f1 unde pf este pointer la int iar &f1 este pointer la pointer

Convenții de tip în cazul de întoarcere parametriiS

Nu trebuie uitat că tipul de date întors de funcție trebuie să fie în coincidență cu tipul de pointer la funcție declarat.

De ex. fie declarațiile:

```
int if1( ) , if2( ) , (*pif)( );
float ff1( ) , (*pff)( );
char cf1( ) , (*pcf)( );
```

şi secvența de asignări :

```
pif = if1;  //legal, tipurile sunt identice
pcf=cf1;   //ilegal, nepotrivire de tip
pff=if2;    //ilegal, nepotrivire de tip
pcf=cf1;   //legal
if1=if2;     //ilegal, se încearcă asignarea unei noi valori la o constantă
```

Apelul unei funcții folosind pointeri

Ex.:

```
int f1( );
int (*pf)( );
int rez;
…
pf=f1;  //am asignat pf la funcție
...
rez= (*pf)(a);   //am apelat funcția f1 cu a ca parametru
```

Obs: (*pf)(a) este ECHIVALENT cu (****pf). Sintaxa ANSI permite ca în loc de (*pf)(a) să folosesc pf(a). Pentru portabilitatea unui program este indicat să se folosească prima formă.

Întoarcerea ca rezultat de pointer la funcțiiS

O funcție poate întoarce un pointer la altă funcție, dar trebuie avut grijă despre modul în care se va face declararea acestei funcții. De exemplu:

```
Int (*f(float x, float y))( )
sau
int (*f(x,y))( )
float x,y;
{
…
}
```

declară o funcție ce are variabile de intrare x,y şi care întoarce un pointer către o funcție ce întoarce un întreg.

Să luăm un exemplu mai concret de folosire a acestor facilități şi anume o funcție de sortare, optimală din punct de vedere al vitezei.

Să analizăm puțin problema. Există mai mulți algoritmi de sortare, fiecare cu avantaje şi dezavantaje:

Quicksort

→ foarte rapid pentru date aleator aranjate

→ ineficient dacă datele sunt parțial sortate pe grupuri

Mergesort

→ unul dintre cei mai eficienți, dar cere memorie

Heapsort

→ cel mai indicat pentru arii masive de date, deoarece necesită cea mai puțină memorie

Pp. că în urma unor analize putem decide care dintre cei trei algoritmi vor fi utilizați. Poate fi scrisă următoarea funcție:

```
Void (*my_sort (float in_data)( )
        {
        extern void quick_sort( ) , merge_sort ( ) , heap_sort ( );
        // heap_sort poate lipsi
        // se realizează analiza datelor . Să presupunem că decizia va fi
        //depusă într-un flag
        switch ( flag )
                case 1: return quick_sort;
                case 2: return merge_sort;
                case 3: return heap_sort;
                default: puts(" Error unknown ");
                        exit(1) break;
        }
```

la apel (my_sort (list)) (list);

Recursivitatea

O funcție recursivă este una care se autoapelează. De ex.:

```
void recurs (void)
{
static cont=1;
printf("%d \n ", cont);
cont++;
recur( );
}
void main (void)
{
recur();
}
```

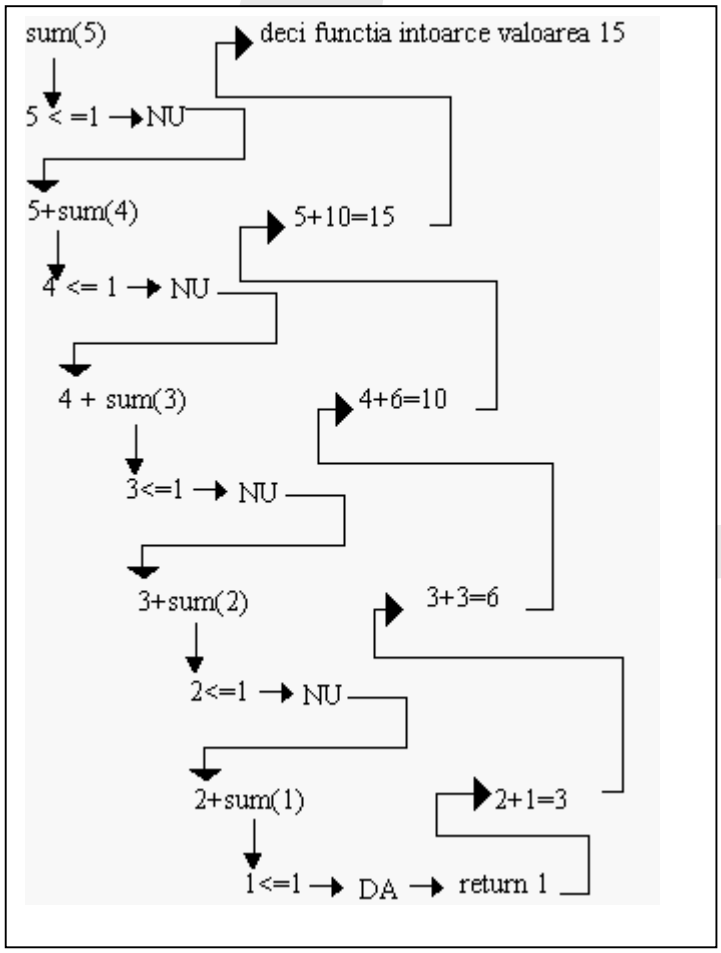
```
void recur (void)
{
static cont=1;
if (cont > 3 ) return;
else
{
printf(" %d \n ", cont);
cont++;
recur( );
}
}
```

La fiecare apel de funcție în stivă sunt salvați o serie de parametrii. Programul de mai sus va autoapela funcția la ∞(1,2,3...), dar stiva PC este finită, având ca rezultat o blocare generală. De aceea orice funcție recursivă trebuie să aibă o condiție, numită caz de bază, care să permită stoparea acestui proces. Un exemplu simplu ar fi în cazul nostru limitarea contorului. Deci:

Întrebări:

1.Ce se întâmplă dacă ştergem else şi cele două acolade de după el ?

2.Ce se întâmplă dacă cont nu e declarat static ?



Folosirea variabilelor fixe este o cale de a controla recursivitatea , după cum am văzut anterior. În exemplul de mai jos se calculează $\sum_{i=1}^{n} i$, i∈ N, recursiv.

```
Int sum(int n)
{
if (n<=1) return n;
else return (n+sum (n-1));
}
```

Pt. n=5 să vedem cum se execută pas cu pas

(Clase de stocare) Storage classes

Majoritatea aplicațiilor mari sunt scrise de echipe de programatori. După proiectarea (schițarea) aplicației în comun fiecare programator pleacă şi realizează o bucată izolată de program. Izolarea se referă la faptul că la realizarea acelui program nu se ține cont de ceilalți, ci la funcțiile care trebuie să le îndeplinească programul. La sfârşit aceste programe sunt unite (linked) împreună pentru a forma aplicația dorită.

Pentru ca în această fază să nu apară probleme majore trebuie să existe un mecanism ce asigură ca variabilele declarate de un programator să nu intre în conflict cu variabilele cu acelaşi nume folosite de un alt programator (de ex. variabilele de contor ale unei bucle: i,j,k). Pe de altă parte este necesar ca anumite date să fie comune ,vizibile şi/sau modificabile de către mai multe surse ce provin de la autori diferiți. Deci trebuie să existe un mecanism care să asigure că numele variabilei declarate în diverse fişiere se referă la aceleaşi locații memorie.

În C se defineşte

→ Cand trebuie să fie o variabilă comună sau împarțită (shared)

→ În ce parte de cod este valabil acest lucru prin atribuirea unui scop (scope) unei declarații

"Scope" este un termen tehnic care denotă regiunea textului sursei C în care numele (declarat) al unei variabile este activ(există).

Altă proprietate a variabilelor este durata care se referă la (descrie) durata de viață în memorie a unei variabile.

Variabilele cu durată fixă sunt garantate că-şi rețin datele chiar după ce şi-au îndeplinit scopul.

În schimb în cazul variabilelor cu durată automată acest lucru nu mai poate fi garantat.

Definiție: **Prin clasă de stocare (storage class) se înțelege scopul şi durata unei variabile.**

Fie următorul segment de program:

```
void f1(void)
{int j;
static int vect[]={1,2,3,4};
…
}
```

Se observă că există două variabile: j şi vect. Ele au un scop de bloc deoarece sunt declarate în interiorul acestuia. Variabilele cu scop de bloc sunt de obicei numite variabile locale. Implicit, variabilele locale au durata automată. În cazul lui j, el va avea atribuită o nouă adresă de memorie la fiecare execuție a respectivului bloc. În cazul lui vect datorită folosirii cuvântului cheie static atribuirea unei adrese de memorie se face o singură dată şi se păstrează pe toată durata execuției programului.

Durata fixă față de cea automată

După cum o precizează şi numele, o variabilă fixă este una staționară, în timp ce pentru o variabilă automatică, zona de stocare în memorie este aleasă automat în timpul execuției programului. Deci o variabilă fixă are memorie alocată, memoria la începerea execuției programului şi-şi păstrează această alocare pe toată durata execuției acestuia.

Variabila automatică se referă la zona de memorie ataşată ei doar atâta timp cât codul în care-şi are scopul este executat. O dată încheiat scopul(zona unde era fixată) compilatorul asigură eliberarea zonei de memorie ocupată de aceasta.

Variabilele locale (cu scopul limitat la nivel de bloc) sunt implicit automatice. Ele pot fi forțate să lucreze static prin folosirea cuvântului cheie static.

Inițializarea variabilelorS

Diferența dintre variabilele fixe şi cele automate este importantă în special pentru variabilele neinițializate. În cazul celor automate, ele vor fi reinițializate la fiecare trecere prin zona lor de existență.

```
void mc(void)
{
        int j=1;
        static int k=1;
        j++;k++;
        printf("j=%d \t k=%d \n",j,k);
}

void main(void)
{
        mc();                    j=2     k=2
        mc();                    j=2     k=3
        mc();                    j=2     k=4
}
```

Altă diferență apare la inițializarea implicită a variabilelor:

→ variabilele automate nu sunt inițializate
→ variabilele statice sunt implicit inițializate cu 0

Ex:

```
void mc1(void)                          void main(void)
    {                                   {
        int j;                              mc();
        static int k;                       mc();
        j++;k++;                            mc();
        printf("j=%d \t k=%d \n",j,k);  }
    }
```

Cu rezultatul:

```
j=3604481     k=1
j=3265349     k=2
j=3456645     k=3
```

Altă diferență între variabilele cu durată fixă şi cele automate se referă la tipurile de expresii ce pot fi folosite în utilizare.

De ex. următoarele declarații sunt legale:

```
int j=0;k=1;
int m=j+k;
float x=3.141*2.3;
```

în timp ce următoarele nu, deoarece j şi k apar în expresii înainte de a fi declarate:

```
int m=j+k;
int j=0;k=1;
```

Regulile de inițializare a variabilelor cu durată fixă sunt foarte stricte. Inițializarea se poate face numai cu o expresie constantă (deci NU poate conține nume de variabilă).

Folosirea variabilelor cu durată fixă

Un exemplu clasic de folosire a acestora este contorizarea numărului de execuții ale unei funcții pentru a-i putea modifica execuția la intervale regulate.

De ex. să presupunem că avem un program care formatează un text citit dintr-un fişier şi scrie rezultatul în alt fişier. Una din funcții este este print_header () apelată la începutul fiecărui program nou. Eventual aceasta poate fi diferit funcție de paritatea/imparitatea numărului de pagini.

```
#define ODD  0
#define EVEN 1
print_header (char* chop_title)
        {
        static char page_type =ODD;
        if (page_type==ODD)
                {
                printf("\t\t\t\t %s\n\n",chop_title);
                page_type=EVEN;
                }
        else
                {printf("%s\n\n",chop_title);
                page_type=ODD;
                }
        }
```

"Scopul" unei variabile (Scope)

În cadrul proiectării unui program trebuie ținut cont de faptul că există 4 tipuri de regiuni fiecare cu un scop bine definit, în care se poate accesa o variabilă prin nume şi anume: program, funcție, bloc, fişier.

La nivel de program (program scope) înseamnă că variabila este activă între diferitele fişiere sursă ce corespund programului executabil. Variabilele de acest tip sunt cunoscute de obicei sub numele de variabile globale.

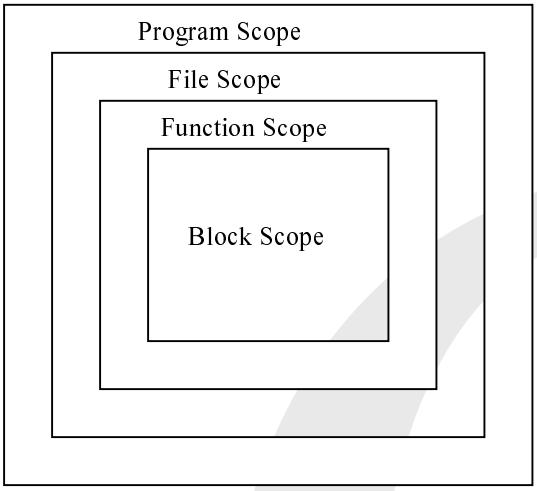
La nivel de fişier (file scope) se referă la faptul că respectiva variabilă este activă de la punctul de declarații până la sfârşitul fişierului sursă ce o conține.

La nivel de funcție (function scope) variabila este activă doar în cadrul funcției.

Introducere in programare

La nivel de bloc de instrucțiuni (block scope) variabila e activă de la punctul de declarație până la delimitatorul de sfârşit de bloc.

În general scopul unei variabile este determinat de locul în care se realizează declararea acesteia.



Ex.:

```
int I;          /* Program Scope */
static int j;   /* File Scope */
func(int k)     /* Program  Scope */
{int m;         /* Block Scope */
start:          /* Function Scope */
…
```

Această ierarhizare permite ca în două funcții diferite să avem variabile cu acelaşi nume sau de ex.:

```
int j=10;
main()
        {
        int j;
        for (j=0;j<5;++j)
                printf("j=%d\n",j);          lucrează normal dar dpdv al altui
        }                                    fişier sursă j=10;
```

10.5. Variabile globale (program sau file scope)

Până acum s-a discutat numai despre variabile locale. Există totuşi cazuri când se doreşte ca o valoare depozitată într-o variabilă să fie vizibilă sau modificabilă din interiorul oricărei funcții. Pentru aceasta este suficient ca respectivele variabile să fie declarate în afara corpului oricărei funcții şi ele vor căpăta astfel un caracter global. Acest tip de variabile sunt automat inițializate cu valoarea zero.

În general în programarea structurată se recomandă evitarea excesului de variabile globale, întrucât nu sunt neapărat necesare decât eventual ca să scriem mai repede şi uşor un cod, dar acest cod nu este neapărat cel mai bun sau cel mai stabil. Mai jos vom prezenta un exemplu simplu relativ la folosirea variabilelor globale.

```
#include <stdio.h>
#include <conio.h>

int global_v1;

int ori_doi(int x);

void main(void)
{
int local_v1;
clrscr();
printf("Variabila globala are valoarea %d\n", global_v1);
printf("Variabila locala are valoarea %d\n", global_v1);
local_v1=ori_doi(local_v1);
printf("Variabila globala are valoarea %d\n", global_v1);
printf("Variabila locala are valoarea %d\n", global_v1);
}

int ori_doi(int x)
  {
   int retval;
   retval = x * 2;
 global_v++;
 return retval;
  }
```

Tematica claselor de stocare ANSI

locul unde se declară / Specificator de clasă	În afara funcției	în interiorul funcției	argumente funcție
auto sau register	NEPERMIS	Scop:block Durata :automatic	Scop: bloc Durata:automatic
Static	Scop:fişier Durata:fixă	Scop: bloc Durata: fixă	NEPERMIS
Extern	Scop: program Durata: fixă	Scop: bloc Durata: fixă	NEPERMIS
Nici un specificator prezent	Scop: program Durata: fixă	Scop: bloc Durata: dinamică	Scop: bloc Durata: automatică

10.6. Macrouri

În mod normal când o funcție este apelată, se creează setul de variabile locale, apoi acestea se inițializează. Acest lucru durează, în unele cazuri, mai mult decât execuția corpului funcției. Apare întrebarea cum poate evita pierderea de viteză şi totuşi să rezulte un program structurat.

Limbajul C este prevăzut cu macrouri (macrodefiniții) care permit implementarea rapidă de operații simple fără dezavantajele de timp din cazul funcțiilor.

Dacă se doreşte declararea unei valori constante se poate folosi directiva preprocesor #define, ca mai jos:

#define pi 3.1416

La întâlnirea respectivei directive preprocesorul va înlocui fiecare menționare în cod a lui pi cu însăşi valoarea 3.1416. Similar se va întâmpla şi cu corpul unui macro.

Observație: o variabilă poate fi declarată fi declarată ca fiind statică, adică nu-şi va modifica valoarea pe parcursul execuție cu ajutorul cuvântului cheie static pus înaintea specificatorului de tip.

Fie exemplele de mai jos unde sunt prezentate două macrouri, unul calculează x^2 iar altul realizează maximum dintre două numere.

#define my_max(x,y) (((x)>(y))?(x):(y))
#define my_sqr(x) ((x)(x))*

Dacă se vor folosi aceste macrouri, în locul apelului unei funcții, se va insera direct codul corespunzător implementării definițiilor din dreapta.

Totuşi folosirea macrourilor ascunde un mare pericol pentru cineva care nu are suficientă experiență. Dacă în cazul unei funcții se realizează întâi evaluarea parametrilor, la apelul unui macro acest lucru nu se mai face. Apar probleme de ordine a execuției operațiilor ca în exemplele de mai jos:

*rez=my_sqr(x+1) // evaluarea este rez=x+1*x+1 în loc de*
//(x+1)(x+1)*

sau

*rez=z/my_sqr(x) //evaluarea este rez = (z/x)*x în loc de z/(x*x)*

Din acest motiv la compilatoarele de OOP s-a introdus noțiunea de funcție "in-line" care elimină dezavantajele macrourilor dar se comportă ca acestea.

Trebuie menționat că la depanarea pas cu pas se poate intra în interiorul unui apel de macro ceea ce face ca erorile de tipul prezentat mai sus să fie greu depistabile.

10.7 Macrouri predefinite

10.7.1. ASSERT

Folosirea acestei macrodefiniții este indicată din următoarele motive:

- creşte foarte mult viteza de depanare a unui program
- programarea în C la ora actuală este realizată sub compilatoare de C++ care acceptă şi cod C clasic, compilatoare care o au predefinită.

Este un macro care funcționează exact ca o instrucțiune if dar, spre deosebirea de aceasta, în momentul în care condiția de evaluat ia valoarea zero blochează execuția programului apelând funcția abort. La stoparea execuției se trimite către dispozitivul standard de ieşire un mesaj specific care precizează numele fişierului sursă şi linia din cadrul acestuia, care a generat eroarea.

Dacă valoarea evaluată este diferită de zero assert nu are nici o influență asupra execuției programului. Prototipul este definit în biblioteca ASSERT.H şi arată ca mai jos:

void assert (int test).

De obicei este folosit pentru a testa dacă o încercare de alocare dinamică a unei noi zone de memorie a reuşit sau nu. După cum am spus el este folosit în cadrul depanării unei aplicații. O dată terminată această fază se poate face ca preprocesorul C să ignore liniile din codul sursă ce conțin apeluri de acest tip prin includerea în codul sursă a directivei

#define NDEBUG

înaintea directivei ce include biblioteca ASSERT.H. Un exemplu scurt de folosire este prezentat mai jos:

```
// aici se va include #define NDEBUG
#include <assert.h>
#include <alloc.h>

void main(void)
    {
    char *s;

    s=( char *)malloc(20*sizeof(char));
    assert(s);
    s=NULL;
    assert(s);
    }
```

La execuția acestui program se va obține activarea assert de-abia în ce-a de-a doua situație.

10.7.2. Macrouri specializate în clasificarea caracterelor.

Au fost introduse pentru a uşura gândirea şi scrierea codului în cazul în care se fac prelucrări pe şiruri de caractere. Aceste prelucrări sunt deseori întâlnite. S-a preferat implementarea ca macrouri pentru a obține o viteză maximă de execuție. Vom prezenta mai jos cele mai folosite macrouri puse la dispoziție de biblioteca CTYPE.H. Toate aceste macrouri returnează o valoare diferită de zero în cazul în care sirul de caractere evaluat are respectiva caracteristică testată de macrou sau zero în caz contrar.

int islower(int c);

Testează dacă caracterul c este din categoria literelor mici , c∈[a…z].

int isalpha(int c);

Testează dacă caracterul c este literă, c∈[a…z] sau c∈[A…Z].

int isupper(int c);

Testează dacă caracterul c este literă mare c∈[A…Z].

int isprint(int c);

Testează dacă caracterul c este tipăribil c∈[0x20…0x7e]

int isascii(int c);

Testează dacă caracterul octetul low al lui c exte în intervalul [1…127].

int ispunct(int c);

Testează dacă caracterul c este un caracter de punctuație.

int isspace(int c);

Testează dacă caracterul c este spațiu, tab, CR, salt la linie nouă , tab vertical sau formfeed adică c∈[0x09…0x0d, 0x20]

int isdigit(int c);

Testează dacă caracterul c este cifră, c∈[0…9].

int isgraph(int c);

Similară cu isprint numai că se elimină spațiile.

int isxdigit(int c);

Testează dacă caracterul c aparține bazei 16, c∈[0…9] sau c∈[a…f] sau c∈[A…F].

10.7. 3. Macrodefiniții predefinite

Unul din următoarele simboluri este predefinit, în funcție de modelul de memorie ales pentru compilare:

__COMPACT__ __MEDIUM__ __HUGE__

__LARGE__ __SMALL__ __TINY__

Alte macrodefiniții implicite:

__cplusplus definit dacă se compilează un program C++
__DATE__ data compilării, ca "mmm dd yyyy"
__FILE__ numele fişierului sursă curent (afectat de #line), ca un şir de caractere
__LINE__ numărul liniei curente (afectat de #line), ca un număr întreg
__MSDOS__ întotdeauna definit (1)
__OVERLAY__ #definit dacă este activată opțiunea de lucru cu reacoperiri (opțiunea -Y activă)
__PASCAL__ #definit dacă opțiunea -p este activă
__STDC__ #definit dacă se cere compatibilitate cu standardul ANSI (opțiunea -A, sau din mediul integrat)
__TIME__ ora curentă "hh:mm:ss"
__TURBOC__ versiunea compilatorului, ca o constantă hexazecimală.

11. Pointeri şi alocarea dinamică

11.1. Introducere.

Primele minisisteme cu microprocesoare pe 8 biţi aveau 64 ko memorie RAM. Această dimensiune era aleasă din cauza unei limitări datorate numărului n de biţi necesari pentru construirea adresei $2^{16} = 2^{10} * 2^{6} = 2^{6}$ ko = 64 ko deci 16 biţi de adresă. Întrucât procesorul lucra pe 8 biţi pentru a avea acces la o dată oarecare din memorie el trebuia să facă una sau două operaţii (depinzând de construcţia internă). Sistemele cu procesoare pe 16 biţi (8086 /V24 /8088) aveau un nou set de instrucţiuni, erau mai rapide şi accesau mai multă memorie RAM .

Problema era că deja se cheltuise foarte mult timp şi bani pentru a dezvolta programe pentru sistemele anterioare. Multe aplicaţii mai importante trebuiau rescrise cu un preţ minim. În toate programele se ţinuse cont în lucrul cu datele din memorie de limitarea de 64 ko.

Cel mai mic efort de programare se făcea dacă se proiecta un sistem de operare care să "vadă " memoria RAM ca fiind alcătuită din bucăţi (segmente) de câte 64 ko. Aceasta a fost idea de bază a sistemului de operare MS-DOS unde adresa de memorie a oricărei date este compusă din două părţi :

- adresa de segment: arată a câta locaţie din cele 2^{16} locaţii ale unui segment este data utilă;
- adresa de offset: arată în al câtelea segment , din cele *n* segmente din memorie, ne aflăm, unde:

$$n = \frac{\dim mem.RAM[ko]}{64[ko]} .$$

În programarea de nivel înalt (la acea dată) vezi C, Pascal, Basic, Cobol ş.a.m.d., actual numite limbaje de nivel mediu, s-a introdus noţiunea de pointer. Aceasta este o variabilă care în loc să reţină o dată utilă direct în calcule, reţine adresa în memorie a unei date utile.

11.1.1. Noţiunea de pointer

Una din cele mai "puternice" modalităţi de acces la date se bazează pe posibilitatea de a avea acces direct la adresa de memorie a unei date stocate acolo.
O variabilă care stochează adresa altei variabile se numeşte pointer.

Avantajul folosirii de către programator a unui asemenea concept este acela că în cazul manipulării unor masive de date de mari, care la rândul lor conţin elemente de mari dimensiuni (vezi cazul unei structuri) nu se va mai transfera toată zona de memorie ocupată de o dată ci numai cu adresa ei de început. Acestea se vor observa şi înţelege mai clar din exemple.

Pentru a vedea mai clar folosirea conceptului de pointer să considerăm programul de mai jos. Acest program declară un vector de elemente întregi (deci reprezentate pe 2 octeți fiecare) precum şi un pointer la un întreg. Aceasta înseamnă că compilatorul va şti în cazul manipulărilor matematice cu adresa care o conține să facă calculele corect, după cum vom vedea mai jos.

Geneza operatorului &op

Se traduce prin "adresa lui" aplicată unui operator op; este o moştenire din vremea DECPDP11 care folosea acest sistem.

Este neplăcut că terminologia s-a păstrat , deoarece ambiguitatea apare în faptul că nu ştim concret despre ce adresă este vorba. La ce ne referim: cea ce poate reprezenta o adresare perfect normală pentru procesorul X poate să fie imposibilă pentru controllerul produs de Y pentru o maşină de spălat care foloseşte adresarea de 17 biți . Deci problema portabilității absolute s-a păstrat pentru că a inventa un nou termen ar fi bulversat mai mult situația.

Un pointer este folositor dacă este o cale de acces la valoarea a cărei adresa o reține.

Limbajul C foloseşte pentru aceasta operatorul *.

```
/********începe zona includeri de bilioteci ***************/
#include <stdio.h>
#include <conio.h>
/*********sfârşit  zona includeri de bilioteci ***************/
// nu am funcții proprii
// nu am variabile globale
/********* incepe zona programului principal**************/
void main(void)
    {0
    int *p; //pointer la o variabilă de tip întreg0
    int v[20]; // tablou de 20 elemente de tip întreg0
    int n,i; //variabile necesare prelucrării masivului de date0
    clrscr();// şterg ecran0
    printf("dati dimensiunea vectorului N=");0
    scanf("%d",&n);//citesc dimensiunea vectorului0
    for(i=0;i<n;i++)0
        {0
        printf("\nV[%d]=",i);0
        scanf("%d",&v[i]);0
        }// citesc vectorul0
    p=v;// depun adresa de început a masivului v în variabila p
    for(i=0;i<n;i++)0
        {0
        printf("\nContinutul lui *v+%d=%d",i,*(v+i));0
        printf("\nAdresa lui V[%d] este %p",i,v+i);0
        }0
    for(i=0;i<n;i++)0
        {0
```

```
        printf("\nContinutul lui *p+%d=%d",i,*(v+i));0
        printf("\nAdresa lui p+%d este %p",i,p+i);0
        }0
    getch();
    }
/*******sfârşit zona program principal*****************/
// neavând funcții proprii nu am aici nimic
```

Ieşirea acestui program în cazul execuției pentru un vector de trei elemente arată ca mai jos:

```
dati dimensiunea N=30
V[0]=10
V[1]=20
V[2]=30
Continutul lui (v+0)=10
Adresa lui V[0] este 0C5E:0FD0
Continutul lui *v+1=20
Adresa lui V[1] este 0C5E:0FD20
Continutul lui *v+2=30
Adresa lui V[2] este 0C5E:0FD40
Continutul lui *p+0=10
Adresa lui p+0 este 0C5E:0FD0
Continutul lui *p+1=20
Adresa lui p+1 este 0C5E:0FD20
Continutul lui *p+2=30
Adresa lui p+2 este 0C5E:0FD4.0
```

11.1.2. Operații aritmetice cu pointeri

Operațiile aritmetice permise sunt:

- adunarea unui pointer cu un întreg
- scăderea unui întreg dintr-un pointer
- scăderea a doi pointeri
- compararea unui pointer cu constantă NULL (valoarea întreaga 0)
- compararea a doi pointeri.

Singura operație aritmetică permisă asupra pointerilor către funcții este compararea cu constanta NULL.

Adunarea unui pointer cu un întreg are ca rezultat un pointer de acelaşi tip. În cazul în care pointerul are ca valoare adresa unui element de tablou rezultatul adunării este adresa elementului al cărui indice este egal cu suma dintre indicele elementului către care indică pointerul, plus valoarea întregului.

Din motive legate de modul de generare a adreselor fizice de către procesor, rezultatul nu este valid decât dacă indicele rezultat din calcul este mai mare sau egal cu zero şi mai mic sau egal cu numărul de elemente din tablou.

ScăderealSunui întreg dintr-un pointer. Se face ca şi cum s-ar aduna la pointer valoarea întreagă respectivă cu semn schimbat.

Scăderea a doi pointeri. Cei doi pointeri trebuie să indice către două elemente din acelaşi tablou; rezultatul este un întreg egal cu diferența indicilor celor doua elemente.

Compararea a doi pointeri. Cei doi pointeri trebuie să indice către doua elemente din acelaşi tablou; rezultatul comparației este egal cu rezultatul comparației indicilor celor doua elemente de tablou.

Valorile aparținând unui tip pointer pot fi convertite către orice alt tip pointer prin folosirea unei conversii explicite.

11.1.3 Modele de memorie

Structura procesoarelor 80x86 a determinat maniere diferite de generare a codului de către compilatoare, in funcție de modurile de adresare utilizate
pentru accesul la date, sau de modul de realizare a apelurilor de funcții.

Dacă programatorul cunoaşte complexitatea aplicatiei si volumul de date manipulat, el poate impune o anumita modalitate de generare a codului, astfel incit el sa fie optim. Aşa a apărut noțiunea de "model de memorie" specific majoritarii compilatoarelor de pe o maşină compatibilă IBM-PC.

Pentru manipularea datelor se folosesc, adesea, pointeri. În funcție de modurile de adresare utilizate, ei pot avea atributele:

- **near pointer** pe 16 biți, reprezentând, de fapt, un offset in interiorul unui segment;
- **far pointer** pe 32 biți, reprezentând o combinație segment:offset (adică o adresă reală completă); un pointer "far" poate avea orice valoare (in limita a 1 MB de memorie), dar în calcule, segmentul nu poate fi modificat;
- **huge pointer** este similar pointerului "far", dar în calcule segmentul poate fi modificat, astfel incât să poată fi adresată toată memoria, fără schimbarea explicită a segmentului. Această caracteristică este dată de operația de normalizare la care sunt supuşi pointerii "huge" în orice operații de calcul.

Observație: Încălcarea regulilor de mai sus nu este semnalată întotdeauna prompt de compilator. Rezultatele pot fi incerte. Modelele de memorie, utilizate de Turbo C, sunt: TINY, SMALL, MEDIUM, COMPACT, LARGE şi HUGE. Diferențierea lor este dată de modul de utilizare a zonelor de cod, date si stiva, precum şi de atributele implicite ale pointerilor utilizați în modelul respectiv. Vom face distincție între zonele de date statice şi zonele de date alocate dinamic (heap). Ca reguli generale pentru alocarea datelor se ține cont de următoarele:

- datele statice se alocă în segmentul de date (DS) şi sunt limitate la cel mult 64 kB;
- datele alocate dinamic se plasează fie în segmentul de date (la modelele TINY, SMALL, MEDIUM), fie in "heap" (la modelele COMPACT, LARGE, HUGE);
- datele auto se alocă în segmentul de stivă (SS) şi sunt limitate la cel mult 64 kB.

11.2. Alocare dinamică.S

11.2.1. Introducere

Până acum s-a discutat despre ceea ce se numeşte alocare statică . Acest termen se referă la faptul că o dată alocată o variabilă în program, memoria corespunzătoare ei numai poate fi refolosită dacă la un moment dat nu ar fi nevoie de ea pe parcursul execuției programului. Deoarece foarte multe ori memoria RAM era extrem de scumpă, iar necesitățile de memorie foarte mari s-a încercat rezolvarea acestui conflict.

Prima metodă era ca atunci când nu ajungea memoria RAM se folosea o unitate externă de stocare cu posibilități de scriere sau citire. Această metodă este şi acum folosită. Singurul dezavantaj al acestei metode este faptul că operațiile cu discul sunt mult mai lente decât cu memoria RAM.

De aceea s-a pus problema folosirii cât mai optime a acestei memorii. În acest scop proiectanții unui sistem de operare au prevăzut în cadrul acestei funcții specifice de gestionare a memoriei.
Printre facilitățile de bază ale acestor funcții sunt :

- alocare de memorie;
- dealocare de memorie;
- verificarea spațiului de memorie rămas nealocat .

Alocarea memoriei. Prin termenul de alocare se înțelege rezervarea unei zone de memorie de o anumită dimensiune, această zonă de memorie “dispare” practic din zona de memorie în care sistemul citeşte sau scrie fără restricții. La alocare se returnează adresa de început a acestei zone.
Dezalocarea memoriei. Prin termenul de dezalocare se înțelege fenomenul invers alocării. Se trimite către sistem adresa de început a unui bloc alocat (rezervat) şi i se cere eliberarea lui adică reintegrarea în spațiul de memorie ce poate fi accesat de orice alt program.
Verificarea spațiului de memorie nealocat. După un număr oarecare de alocări, fără a mai face şi dezalocări spațiul rămas liber tinde către zero. Funcțiile de alocare în momentul în care li se cere să aloce o anumită zonă (*n*-octeți) şi nu mai au suficient spațiu liber returnează NULL pentru a semnaliza această imposibilitate. Totuşi aceasta reprezintă o situație critică şi nu se mai poate face mare lucru. De aceea, ca tehnică de programare, se poate face din când în când în punctele cheie verificarea spațiului de memorie rămas liber pentru a calcula automat dacă programul poate continua sau nu.

Eroare de gândire a începătorului relativ la operatorul sizeof

Acest operator ştie să întoarcă numărul de octeți pe care este stocat un tip de dată DEFINIT fie implicit (tipurile de dată implicite : char , int , long, double) fie cele definite de utilizator (structurile).

Dar, NU POATE calcula dimensiunea unei zone despre care nu are informații clare în momentul compilării.

Exemplu clasic :

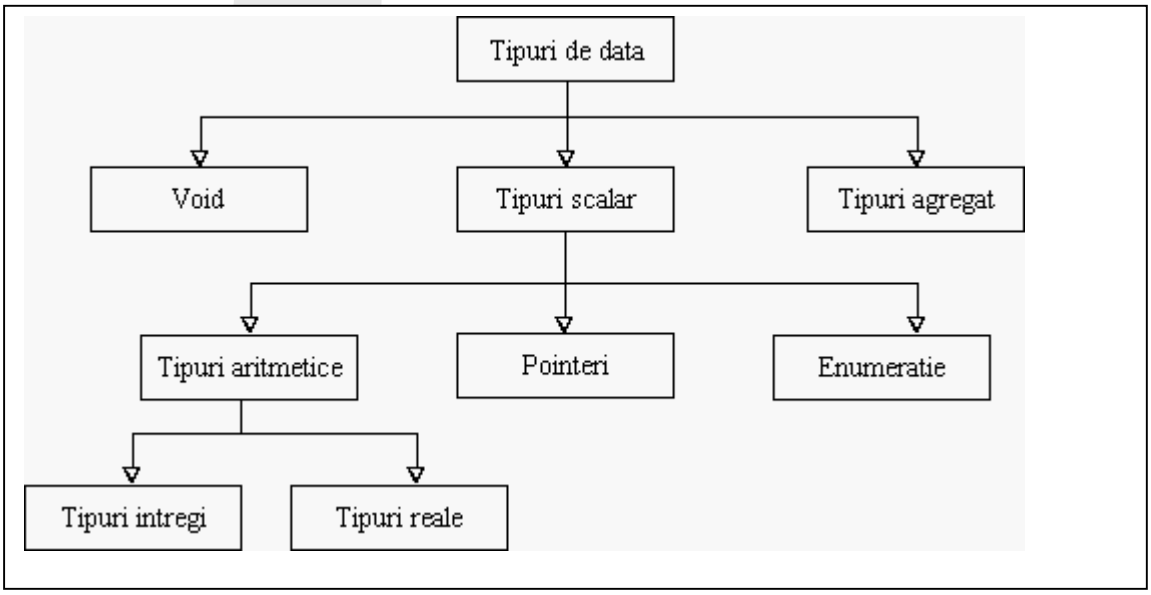
```
char S[10];
 .......
gets(S);
n=sizeof(S)   // eroare
              // S poate avea fie 255 caractere (caz  în care se păstrează sigur numai
              // primele 10, fie nici un caracter
              // deci nu poate fi aplicat.
```

De aceea s-a introdus funcția strlen(S) în acest caz.

O versiune de strlen(S) ar fi

```
   nr_cars=0;
   while ( *S[i++]!='\0') nr_cars++;
   return nr_cars;
```

Mai jos prezentăm o altă clasificare a tipurilor de dateŞ



11.2.2. Alocarea dinamică

Orice tip de dată predefinit sau definit de utilizator poate fi alocat static sau dinamic. Funcția de bază în limbajul C, folosită pentru alocarea dinamică are următorul prototip:

*void *malloc(size_t size);*
*void far *farmalloc(unsigned long nbytes);*

Alocarea dinamică a unui bloc de memorie de lungime dată (size, sau nbytes). "malloc" permite alocarea din segmentul de date a cel mult 64 kB, iar "farmalloc" permite alocarea din "heap" a cel mult 1 MB (de fapt in limitele memoriei ramase disponibile). Funcțiile returnează pointer NULL, dacă nu mai există suficientă memorie pentru alocare.

*void *calloc(size_t nelem, size_t elsize);*
*void far *farcalloc(unsigned long nunits, unsigned long unitsz);*

Alocarea dinamică a unui bloc de memorie pentru un număr dat de elemente (nelem, nunits) de dimensiune cunoscută (elsize, unitsz). Dimensiunea totală alocată este produsul celor doi parametri. "malloc" permite alocarea din segmentul de date a cel mult 64 kB, iar "farmalloc" permite alocarea din "heap" a cel mult 1 MB (de fapt în limitele memoriei rămase disponibile). Funcțiile returneaza pointer NULL, dacă nu mai există suficientă memorie pentru alocare.

*void *realloc(void *block, size_t size);*
*void far *farrealloc(void far *oldblock, unsigned long nbytes);*

Modificarea dimensiunii unui bloc de memorie deja alocat. Pointerul block (oldblock) trebuie să fi fost obținut cu o funcție de alocare, apelată anterior: "malloc" ("farmalloc"), "calloc" ("farcalloc"). Dacă redimensionarea nu se poate face, funcțiile returnează NULL. Este posibil ca noul pointer rezultat să fie diferit de pointerul inițial.

*void free(void *block);*
*void farfree(void far *block);*

Eliberează un bloc de memorie deja alocat. Pointerul block trebuie să fi fost obținut cu o funcție de alocare, apelată anterior: "malloc" ("farmalloc"), "calloc" ("farcalloc"). Încercarea de a apela una din funcțiile de mai sus cu un pointer obținut pe alte căi, poate duce la rezultate imprevizibile.

În programele ce utilizează alocare dinamică, se recomandă ca orice zonă, astfel obtinută, să fie eliberată după utilizare.

Obs. Compilatorul nu sesizează erori de tipul celor menționate mai sus; depanarea este, de asemenea, dificilă.

unsigned coreleft(void);
unsigned long farcoreleft(void);

Evaluează disponibilul de memorie posibil de a fi folosit în alocarea dinamică. Funcția "coreleft" returnează memoria liberă din segmentul de date, iar "farcoreleft" memoria liberă din "heap".

Funcțiile de alocare primesc numărul exact de octeți ce se doresc a fi alocați şi returnează adresa de început a respectivei zone. Din motive de portabilitate a programului nu este indicat să se calculeze dimensiunea zonei de alocat cu formula:

Nr_elemente*.nr_octeți_ocupat_de_respectivul_tip_dată

Şi acest lucru deoarece funcție de arhitectura sistemului de calcul numărul de octeți pe care este reprezentat acelaşi tip de date poate diferit. Deci programul ar trebui modificat în cazul recompilării sub altă platformă.

Portabilitatea unui program

Se referă la modul de scriere a unui cod sursă astfel încât la recompilarea sau la execuția lui sub diferite platforme hardware sau software, modificările ce trebuie făcute pentru a-l face funcțional să fie minime sau inexistente.

Pentru a asigura portabilitatea din punct de vedere al alocării dinamice se va folosi un operator special al limbajului C şi anume sizeof(tip_dată). Numele se traduce prin "mărimea lui". El returnează numărul de octeți pe care este reprezentat respectivul tip de dată. În aceste condiții rezultă clar că zona de memorie ce se doreşte a fi alocată trebuie calculată cu următoarea formulă:

Nr_octeți_de_alocat = nr_elemente**sizeof(*tip_dată*)*

Acest operator ştie să facă calculul numărului de octeți ocupați şi în cazul unui tip de dată definit de utilizator.

11.2.3. Alocarea dinamică a tablourilor de date

După cum ştim în cazul tipurilor de date alocate static se pot defini tablouri care să aibă până la trei dimensiuni. Există în respectivul caz o limitare a zonei de memorie care poate fi astfel alocată şi anume între 64 ko şi 1 Mo funcție de modelul de memorie ales pentru compilare. Dacă în cazul declarării statice compilatorul se ocupă de tot procesul în cazul alocării dinamice acest proces devine ceva mai complicat. Există două metode de alocare:

Metoda I

Alocarea se face o singură dată pentru toate elementele şi este o singură instrucțiune. Metoda are, pentru tablourile a căror dimensiuni sunt >= 2, dezavantajul că accesul la un element se poate face numai prin intermediul unui pointer şi trebuie să folosesc o formulă de calcul a adresei elementului. Această formulă trebuie să țină cont de adresa de început a masivului, de indicii elementului şi de numărul de octeți pe care este reprezentat. Aşa se lucra la nivelul limbajului de asamblare unde nu aveam alte posibilități.

Formula de echivalență **ar[i]** ↔ ***(av+i).**

Vom prezenta un scurt exemplu de folosire a primei metode în cazul unui tablou monodimensional de caractere.

```
#include <string.h>
#include <stdio.h>

void main(void)
    {
    char strg[40],*there,one,two;
    int *pt,list[100],index;

      strcpy(strg,"Acesta e un sir de caractere.");

      one = strg[0];                /* one şi two sunt identici ca valoare */
      two = *strg;
      printf("one, two= %c %c\n",one,two);

      one = strg[8];                /* one şi  two sunt identici */
      two = *(strg+8);
      printf(" one, two=%c %c\n",one,two);

      there = strg+10;          /* strg+10 este identic cu  strg[10]  */
      printf("strg[10]= %c\n",strg[10]);
      printf("there= %c\n",*there);

      for (index = 0;index < 100;index++)
         list[index] = index + 100;
      pt = list + 27;
      printf("list[27]= %d\n",list[27]);
      printf("*pt= %d\n",*pt);
    getch();
    }
```

Metoda II

Porneşte de la observația că, compilatorul face automat calculul adresei elementului în şirul de elemente în cazul declarării statice. Se poate folosi şi în cazul alocării dinamice această facilitate a compilatorului cu condiția ca modul în care se realizează alocarea să respecte modalitatea internă folosită de compilator în cazul alocării statice. Acest lucru permite ca, deşi se va aloca dinamic un tablou, ulterior în cod accesul la oricare element să poată fi realizat folosind metoda indiceală tipică cazului alocării statice.

11.2.3.1. Alocarea tablourilor monodimensionale

În acest caz alocarea se face conform formulei de mai jos:

Nume_tablou=(tip_elem_tablou)malloc(nr_elem_tab *sizeof(tip_el_tablou))

Un exemplu de aplicare în program a acestei formule

```
void main(void)
{
int v*;
        ...
v=(int *)malloc(10*sizeof(int));
...0
}0
```

Se observă că valorii returnate de "malloc" îi aplicăm un operator de cast. În mod normal conversia de tip se face automat, totuşi în cazul alocării este indicat să fie făcută explicit.

Vectorul va putea fi folosit conform celor două metode ca mai jos:

1. *(v+4*sizeof(int))=0x32;
2. v[4]=0x32;

Un caz special este cazul alocării dinamice al unui tablou monodimesional de caractere. Orice şir de caractere este prevăzut cu un terminator special '\0'. Aceasta provine din limbaj de asamblare, având în vedere faptul că funcțiile DOS de lucru cu şirurile de caractere sunt gândite să primească adresa de început din zona de memorie şi să prelucreze element cu element şirul până când este întâlnit terminatorul '$'. Dacă inițializarea se va face element cu element în program '\0' trebuie depus explicit. Folosirea funcțiilor de bibliotecă dă avantajul că terminatorul este pus automat de respectivele funcții.

11.2.3.2. Alocarea tablourilor bidimensionale

Formula de echivalență **ar[m][n]** ↔ ***(ar[m]+n)** ↔ ***(*(ar+m)+n)**

Dacă ar este de întregi int ar[m][n] atunci **ar[i][j]** este echivalent cu

***(int*)((char*)ar + (i*a*sizeof(int) + j*sizeof(int))**

Cast-urile trebuiesc puse pentru că se observă că calculele de deplasament, ținând cont de de spațiul ocupat de un element din tablou, au fost făcute explicit.

În acest caz alocarea este mai complicată, însă ulterior există avantajul de a se putea lucra şi prin specificarea poziției elementului.

```
int far **t_alloc(int lin, int col)
    {
    int i;
    register int far **plin, *pdata;

    pdata=(int far *)farcalloc(lin*col,sizeof(int far*));0
    if(pdata==(int far*)NULL)err(20,30,"Nu exista spatiu");0

    plin=(int far**)farcalloc(lin,sizeof(int far*));0
    if(plin==(int far**)NULL)err(20,30,"Nu exista spatiu");0

    for(i=0;i<lin;i++)0
        {0
        plin[i]=pdata;0
        pdata+=col;0
        }0
    return plin;
    }
/*************************************************/
void t_free(int far **pa)
    {0
    farfree(*pa);0
    farfree(pa);0
    }0
```

Exemplul de mai sus ne permite să pot accesa în program un element din tablou în maniera m[i][j] ceea face codul mai clar.

11.2.3.3. Tablouri tridimensionale

Se tratează după aceeaşi gândire ca şi cele bidimensionale, mai jos vom prezenta un exemplu de tranfer de parametru în cazul tablou tridimensional.

fie

```
void main(void)
        {
        int m [5][6][7];
        …
        f2(m, x, y, z);
        }
void f2 ( int *** argm, dim1, dim2, dim3)
        {int x;
        …
```

atunci x=2 + *((int x) argm + i*dim3*dim2 + j*dim2 + k) (echivalent cu argm[i][j])

11.3. Pointeri la structuri

Având în vedere definiția pointerului nu există nimic care să împiedice declararea unor pointeri la o structură. În aceste condiții tipul de date grupă şi student prezentate anterior se vor rescrie ca mai jos.

```
typedef struct Student
    {
    char *name;
    unsigned short int age;
    unsigned char n_lab;
     unsigned char n_ex;
    } stud;

typedef struct Grupa
        {
        int nume;
        int an;
        stud *t;
        }gr;
    gr *p1,*p2;
```

11.3.1. Accesarea membrilor

De această dată se va folosi → în loc de punct, restul regulilor de accesare rămânând identice. Acest nou operator de acces a fost introdus doar pentru a scrie cod cât mai clar deoarece dacă avem *stud *p*; atunci *p→nume* este perfect echivalent cu *(*p).nume* din punct de vedere sintactic.

11.3.2. Alocarea structurilor de date

Trebuie avut grijă ca tot spațiu de care este nevoie să fie alocat. Există două nivele de alocare: primul se referă la alocarea de spațiu pentru noul tip de dată, al doilea nivel se referă la alocarea de spațiu, unde este cazul, pentru membrii noului tip de dată.

Alocarea la primul nivel:

```
Gr *p1;0
p1=(gr *)malloc(25*sizeof(gr));0
```

Nivelul doi de alocare este clar de-abia în faza de inițializare a unei structuri.

11.3.3. Inițializare şi distrugere structuriS

Dacă alocarea dinamică pentru tipurile de date predefinite e relativ simplă în majoritatea cazurilor în cazul structurilor de date trebuie ca programatorul să-şi construiască propriile funcții de inițializare / eliberare a datelor.

Se vor prezenta funcțiile de inițializare pentru un obiect de tip grupa. Se observă că un obiect de tip grupa conține un tablou de obiecte de tip student şi deci trebuie întâi scrisă funcția care inițializează pe acesta.

```
Stud *init_stud(char *name,unsigned short int age,
                    unsigned char n_lab,n_ex)0
    {0
    stud *p;0
    if(!(p=(stud *)malloc(sizeof(stud))))error("Not enough memory");0
    if(!(p->name=(char *)malloc(strlen(name)*sizeof(char))))0
            error("Not enough memory");0
    strcpy(p->name,name);0
    p->age=age;0
    p->n_lab=n_lab;0
    p->n_ex=n_ex;0
    return *p;0
    }0

 void free_stud(stud *p)
    {
    free(p->name);
    free(p);
    }

gr *init_gr(int nume, stud *st, char name, int age, n1,n2)
    {
    gr *p;
    if(!(p=(gr *)malloc(sizeof(gr)))) error("Not enough memory");
    p->nume=nume;
    p->an=an;
    p->stud=st;
    return p;
    }
void err(char *s)
    {
    clrscr();
    puts(s);
    exit(0);
    }
```

După cum se observă cele două funcții alocă câte un element. Alocarea pentru tabloul de studenți al cărui pointer este primit în init_gr nu a fost prezentată, pentru că se foloseşte funcția de alocare a unui student pentru fiecare din elemente şi ele pot fi depuse într-un tablou alocat static sau dinamic sau într-o listă înlănțuită.

12. Declarații complexe în CS

Introducere in programare

Peste un anumit nivel de experiență, există tendința folosirii unor declarații compacte, dar complexe, şi aceasta compilatorul le acceptă. acest lucru conduce la scăderea clarității codului. de aceea recomandăm evitarea pe cât posibil. De exemplu următoarea declarație declară pe x ca fiind un pointer la o funcție care întoarce un pointer către al cincilea element dintr-un tablou de pointeri la întregi. Complicat , NU?

int *(*(*x)()[5];

Mai fos prezentăm o metodă simplă pentru evitarea acestui tip de complicații:

```
typedef int *AP[5] //al 5-lea element dintr-un tablou de pointeri la întreg
typedef AP *FP() // funcția ce va returna un pointer la al 5-lea element al unui
                 tablou de pointeri către întregi
FP *x            // pointer la.....
```

Motivul principal pentru care declarațiile de tipul celor prezentate mai sus sunt aşa complicate în C, este datorat faptului că operatorul pointer { * } este de tip prefix, iar operatorii de funcție şi tablou sunt de tip postfix.

Pentru a descifra declarații complexe, trebuie privit dinăuntru înspre afară, adăugând * în stânga şi ( ) sau [ ] în dreapta.

Descifrarea declarațiilor complexe în CS

Cea mai bună strategie în descifrarea declarațiilor complexe, este să pornim de la numele variabilei şi apoi să adăugăm(să ținem cont de efectul) fiecare parte(operator) ai expresie pornind cu operatorii cei mai apropiați de numele variabilei.

În lipsa parantezelor: se va ține cont întâi de operatorii de tip funcție sau tablou (din dreapta variabilei, deoarece au prioritate mai mare, şi de-abia la sfărşit se va ține cont de efectul operatorilor de tip prioritar)

De exemplu:

char **[ ]; va fi descifrată în următorii paşi:

1. x[ ] este un tablou
2. **[ ] este un tablou de pointeri
3. char *x[ ] este un tablou de pointeri către caractere.

Parantezele pot fi folosite fie pentru claritate, fie pentru schimbare a precedenței. De exemplu:

int (**[ ]) ( ); va fi descompus după cum urmează:

1. x[ ] este un tablou
2. (**[ ]) este un tablou de pointeri
3. (**[ ]) ( ) este un tablou de pointeri către funcții
4. int ((*x [ ]) ( ) este un tablou de pointeri către funcții ce returnează un întreg

Compunerea declarații complexe

Pentru a compune declarații complexe procesul este identic ca la descompunere. De exemplu dacă se doreşte declararea unui pointer către un tablou de pointeri către funcții care întorc pointeri la tablouri de structuri cu numele S, se vor folosi următorii paşi:

1. (*x) este un pointer
2. (*x)[ ] este un pointer la un tablou

3. (*(*x[ ])) este un pointer la un tablou de pointeri
4. (*(*x[ ])) ( )este un pointer la un tablou de pointeri la funcții
5. (*(*(*x[ ])) ( ))este un pointer la un tablou de pointeri la funcții ce întorc pointeri
6. (*(*(*x[ ])) ( )) [ ] este un pointer la un tablou de pointeri la funcții ce întorc pointeri catre tablouri
7. struct S (*(*(*x[ ])) ( )) [ ] este un pointer la un tablou de pointeri la funcții ce întorc pointeri catre tablouri de structuri cu numele S

În scopul uşurării înțelegerii, se va prezenta un tablou cu diverese expresii complexe uzuale în codurile C, şi anumite greşeli comune.

- int i; un întreg
- int* p; pointer la întreg
- int a[ ]; tablou de întregi
- int f( ); funcție ce întoarce un întreg
- int **p; pointer la pointer către întreg
- int (*pa) [ ]; pointer la tabloul de întregi
- int (*pf) ( ); pointer la o funcție ce întoarce un întreg
- int *ap [ ]; un tablou de pointeri către întregi
- int aa [ ] [ ] un tablou de tablou de întregi
- int af [ ] ( ); un tablou de funcții ce întorc întreg NEPERMIS
- int *fp( ); o funcție ce întoarce un pointer la întreg
- int fa ( ) [ ]; o funcție ce întoarce un tablou de întregi NEPERMIS
- int f ( ) ( ) ; o funcție ce întoarce o funcție ce întoarce un int NEPERMIS
- int ***p; pointer la pointer la pointer la int
- int (**ppa) [ ]; pointer la pointer către un tablou de int
- int (**ppf) [ ]; pointer la pointer către o funcție ce întoarce int
- int * (*pap) [ ]; pointer către un tablou de pointeri la întregi
- int (* paa) [ ] [ ]; pointer la un tablou de pointeri ce rețin întregi
- int (* paf) [ ] ( ); pointer la un tablou de funcții ce întorc întregi NEPERMIS
- int *(*pfp) ( ); un pointer la o funcție ce întoarce întreg
- int (* pfa) ( ) [ ]; pointer la o funcție ce întoarce un tablou de întregi
  NEPERMIS
- int faf( ) [ ] ( ); funcție ce întoarce un tablou de funcții ce întorc întregi
  NEPERMIS
- int *ffp( ) ( ); funcție ce întoarce o funcție ce întoarce un pointer la int
  NEPERMIS
- int (* pff) ( ) ( ); un pointer la o funcție ce întoarce o funcție ce întoarce un întreg
  NEPERMIS
- int **app[ ]; un tablou de pointeri la popinteri către întregi
- int (*apa[ ]) [ ]; un tablou de pointeri către un tablou de întregi
- int (*apf[ ]) ( );un tablou de pointeri către funcții ce întorc întregi
- int *aap[ ] [ ];un tablou de tablouri de pointeri la întregi

- int aaa[ ] [ ] [ ]; un tablou de tablouri de tablouri de întregi
- int aaf [ ] [ ] ( ); un tablou de tablouri de funcții ce întorc întregi NEPERMIS
- int *afp[ ] ( ); un tablou de funcții ce întorc pointeri la întregi
- int afa [ ] ( ) [ ]; un tablou de funcții ce întorc tablouri de întregi NEPERMIS
- int aff [ ] ( ) ( ); un tablou de funcții ce întorc funcții care întorc întregi
  NEPERMIS
- int **fpp ( ); o funcție ce întoarce un pointer la pointer de întregi
- int (*fpa ( )) [ ] ; o funcție ce întoarce un pointer către tablouri de întregi
- int (* fpf ( )) ( ); funcție ce întoarce un pointer către o fucție ce întoarce un întreg
- int faa ( ) [ ] [ ]; funcție ce întoarce un tablou de tablouri de întregi
  NEPERMIS
- int *fap ( ) [ ]; funcție ce întoarce un tablou de pointeri către întregi
  NEPERMIS

Câteva tehnici de creştere a vitezei de execuție a unui cod

Propagarea copiei (se pune la asignare)
Fie următoarea secvență de cod:

x=y
z=1.+x
corect este:
x=y
z=1.+y (elimină interdependențele dintre cele două relații)

Reducerea puterii
Ex: y = pow (x,2) -> ia foarte mult timp

Înlocuiți cu y=x*x sau j=k*2 -> ia mult timp în comparație cu adunarea j=k+k sau cu deplasarea la stânga j=k<<2

Redenumirea variabilelor
Câteodată (când se vrea viteză maximă) este mai bine să evităm refolosirea variabilelor
x=y*z
q=r+x+z
x=a+b

A	**b**
x=y*z	x0=y*z
q=r+x+z	q=r+x0+x0
x=a+b	x=a+b

Compilatorul realizează automat aceste lucruri, dar pentru calcule de finețe este bine să facem personal optimizarea codului. Deoarece a) mănăncă timp de calcul iar b) mănăncă memorie, în opțiunile compilatorului există:

1. optimizare pentru viteză (stil b)
2. optimizare pentru dimensiune (stil a)

Eliminarea subexpresiilor comune

```
Ex.   D=C*(A+B)                 t=A+B
      E=(A+B)/2     ----->      D=C*t
      F=sqr(A+B)                E=t/2
                                F=sqr(t)
```

Atenție la ordinea în care se scriu subexpresiile deoarece pt. compilator a+b+c poate fi diferit de B+C+a chiar când se face optimizare automată.

Eliminarea codurilor constante din bucle

```
for (i=0;i<u;i++)
        {A[i]=B[i]+C*D;
        E=G[k];
        }
```

Se observă că C*D este calculată de N ori şi nu este o funcție de (i) . Absurd! La fel şi E = G[k] este un transfer de memorie realizat de n ori şi nu este o funcție de i, deci corect bucla este :

```
temp=c*d;
for(i=0;i<n;i++) A[i]=B[i]+t;
E=G[k];
```

Simplificarea prin inducție

```
Ex. for (i=0;i<n;i++)
        k=i*4+M
```

După o scurtă analiză matematică (temă acasă) rezultă o secvență echivalentă dar se scutesc n înmulțiri.

```
k=m
for(i=0;i<n;i++)  k=k+4;
```

Se foloseşte când referim direct tablouri

```
adresa = adresa_de_baza(A)+(i-1)*sizeof(data)

adresa = adresa_de_baza(A)-sizeof(data)
   adresa = adresa+sizeof(data)
```

dacă aplic şi eliminarea constantelor, codul devine :

```
temp = sizeof(data)
adresa = adresa_de_baza(a)-temp
```

adresa = adresa+temp

Lucrul cu fişierele de date (I/O FILE) Streamuri (streams)

C nu face distincție între diverse dispozitive cum ar fi terminal, ZIP, HDD.
În toate cazurile I/O este realizat prin aceste stream-uri care sunt asociate cu fişiere sau cu dispozitive.

Def: Un **stream** constă dintr-o înşiruire ordonată de octeți. Poate fi gândit cumva ca un vector, numai că în acest caz nu am acces paralel ci serial la fiecare element.

Pentru a realiza accesarea unui stream trebuie declarat un pointer la o structură standard numită FILE definită in stdio.h.
Ea conține:

- Informații despre numele fişierului
- Modul de acces
- Un pointer către următorul caracter din stream

Aceste informații sunt necesare sistemului de operare, din punctul de vedere al programatorului accesul se face prin intermediul unui pointer la această structură numit pointer la fişier (file pointer).

Un program poate avea unul sau mai multe streamuri deschise simultan.

Unul din câmpurile din FILE este indicatorul de poziție în fişier care se referă la următorul octet ce va fi scris / citit în / din fişier.

După fiecare operație R/W S.O modifică automat acest pointer. Accesarea se poate face direct dar acest caz particular, de care trebuie ținut seama, va face ca un program să nu mai fie portabil.

Atenție: A nu se confunda pointerul la fişier cu indicatorul de poziție !!! Unul este un pointer altul este un deplasament.

Streamuri standard

Există 3 streamuri care sunt deschise automat pentru fiecare program. Ele se numesc stdin, stdout, stderr. În mod normal ele se referă la terminal dar multe sisteme de operare permit redirectarea lor.
Ex.: un mesaj de eroare poate fi scris într-un fişier în loc de terminal; printf() şi scanf() folosesc streamurile stdin şi stdout. Aceste funții pot fi folosite pentru lucrul cu fişierele prin redirectarea stdin şi stdout către respectivele fişiere folosind _____ . Dar mai comod este folosirea funcțiilor fscanf() şi fprintf().

Format text şi format binarS

Datele pot fi accesate în unul din cele două formate standard ANSI –text sau binar. Un stream de tip text constă dintr-o serie de linii, unde fiecare linie este terminată cu un caracter de salt la linie nouă '\n'.
Orice S.O poate avea altă politică de stocare a liniilor pe disc deci terminatorul '\n' nu este obligatoriu. Dar din punct de vedere al programatorului este ca mai sus, restul fiind ascuns de compilatorul C. Totuşi lucrul în mod text nu garantează o portabilitate totală a

unui program. În formatul binar compilatorul nu realizează nici o interpretare a biților ; îi citeşte sau scrie exact cum apar.

Introducere în programare

Bufferarea (folosirea unei zone de memorie ca tampon)

Deoarece comparativ cu memoria, dispozitivele de stocare în masă sunt extrem de încete se încearcă reducerea numărului de operații de I/O cu discul. Bufferarea este cea mai simplă metodă pentru a realiza aceasta.

Un buffer este o zonă unde sunt stocate temporar datele înainte de a fi trimise către destinația lor finală. Toate S.O folosesc buffere pentru a citi/scrie din dispozitivele I/O numai că, în acest caz, bufferul are dimensiune fixă. De obicei, este de 512 sau 1024 octeți. Acest lucru înseamnă că, chiar dacă se doreşte citirea unui singur caracter dintr-un fişier, S.O citeşte tot blocul care conține respectivul caracter.

Biblioteca C conține un strat adițional de bufferare care apare în 2 forme:

- buffere de linie
- buffere de bloc

La bufferarea în linie, sistemul stochează caracterele până la primul '\n' sau până se umple bufferul şi apoi trimite întreaga linie la S.O pentru procesare. Acest lucru se întămplă şi când citesc date de la terminal.

La bufferarea în bloc, sistemul stochează caracterele până când acel bloc este umplut.

Implicit streamurile I/O relativ la fişiere sunt bufferate în bloc, iar cele care se referă la terminal(stdin, stdout) sunt bufferate în linie.

De asemenea biblioteca C include şi un manager de buffer care ține acel buffer cât mai mult posibil în memorie, deci dacă se accesează aceeaşi porțiune dintr-un stream de mai multe ori, există mari şanse ca să se evite accesul la dispozitivul de stocare.

Atenție! Acest lucru poate crea probleme de consistență, dacă fişierul este folosit în comun de mai multe procese. Pentru sincronizarea între procese este nevoie de:

- scrierea în ASM a propriilor _funcții
- folosirea funcțiilor specifice S.O pe care se lucrează

În ambele moduri de bufferare, bufferul poate fi golit la cerere cu ajutorul lui fflush() trimițând, astfel , datele din el la destinația lor.

Deşi acest mod de bufferare este mai eficient decât procesarea individuală a fiecărui caracter, există cazuri în care ne deranjează , şi anume când se doreşte explicit procesarea instantanee a unui caracter fie la intrare fie la ieşire. Cel mai simplu exemplu este atunci când se doreşte procesarea caracterelor introduse de la KBD fără a mai aştepta '\n'.

C permite modificarea memoriei bufferului chiar la acest nivel,tocmai pentru a putea rezolva şi problemele de acest tip.

Citirea şi scrierea datelorS

O dată ce s-a deschis un fişier se va folosi pointerul la fişier pentru a putea realiza operații de scriere/citire. Există 3 nivele de granularitate (finețe):

1. la nivel de caracter
   - câte un caracter odată
2. la nivel de linie
   - câte o linie odată

3. la nivel de bloc
   – câte un bloc odată

O regulă care se aplică la toate nivelele de I/O este aceea că nu pot citi dintr-un stream şi apoi scriu în el fără a apela fseek() , rewind() sau fflush(). La fel şi în cazul invers. Acest lucru este necesar pentru că trebuie golit bufferul înaintea oricărei operațiii pentru a evita problemele de inconsistență.

1. Lucrul la nivel de caracter

getc() - macro , citeşte un caracter
putc() - macro , scrie un caracter
de la stream-ul implicit

fgetc()
fputc()
identice cu anterioarele lor dar sunt fincții

Deşi primele sunt macrouri deci mult mai rapide pot exista probleme de ex:

putc('x', fp[j++]); s-ar putea să nu meargă corect.

De aceea este indicat că atunci când se lucrează ca în exemplu să folosim fputc() sau fgetc();

2. La nivel de linie

Funcții:fgets() şi fputs() şi se specifică parametri
s -> pointer la tabloul de caractere
n -> nr. Maxim de caractere ce vor fi procesate
stream -> streamul cu care se lucrează

3. La nivel de bloc

Un bloc poate fi gândit ca un tablou . Când citim/scriem în bloc trebuie să specificăm atât numărul de elemente al blocului cât şi dimensiunea fiecărui element.

Funcții: fread() şi fwrite() cu următorii parametri:
ptr -> pointer către tabloul ce stochează caracterele
size -> mărime(nr. de octeți al tipului de dată folosit, recomand sizeof() pentru portabilitatea programului
număr elemente ce trebuie procesat
pointer la stream

I/O Nebufferat

Anterior am menționat faptul că există situații în care bufferarea datelor nu deranjează. Exemplu clasic este editorul de text, când se scrie un text şi se apasă tastele de navigare prin el.

Pentru a elimina buferarea pentru stdin se poate folosi fie setbuf(), fie setvbuf(). Setbuf(stdin,NULL) sau stat= setvbuf (stdin, NULL, IONBUF, O).

Accesul aleator într-un fişierS

Este asigurat de funcția fseek care poziționează pointerul la fişier la un deplasament dorit față de începutul acestuia.

Ftell returnează valoarea pointerului de fişier.

12. Facilități de nivel scăzut ale limbajului C

Rutinele utilizate de sistemul de operare MS-DOS pentru tratarea operațiilor de intrare sau ieşire şi gestiunea resurselor sistemului pot fi apelate şi dintr-un program scris intr-un limbaj de nivel înalt cum este C. Limbajul permite şi legarea cu module scrise in limbaj de asamblare.

Rutinele DOS sunt apelate prin întreruperi software. Întreruperea 21h (INT 21h) este serviciul pentru apelul funcțiilor DOS. Pentru apelul unei funcții aceste registre (nu toate) se pregătesc cu diverse valori:

AH conține numărul funcției
AL conține numărul subfuncției (dacă este cazul)

Celelalte registre se încarcă diferit de la caz la caz. Se generează, apoi, întreruperea INT 21H.

Similar cu funcțiile DOS se pot realiza şi apeluri de funcții BIOS pregătind registrele intr-un mod convenabil. Sunt astfel accesibile majoritatea facilităților pe care programatorul le avea numai la nivelul limbajului de asamblare.

Vom descrie in continuare câteva din funcțiile cuprinse in biblioteca C ale căror prototipuri se afla in fisierele dos.h, bios.h, setjmp.h, process.h.

12.1. Funcții pentru acces direct la memorie

```
unsigned FP_OFF(farpointer);
unsigned FP_SEG(farpointer);
void far *MK_FP(unsigned seg, unsigned ofs);
```

Aceste trei macrouri sunt folosite pentru a returna:

-offsetul unui pointer far;
-segmentul unui pointer far;
-un pointer far cu segmentul seg si ofsetul ofs.

Exemplu:

```
#include <dos.h>
#include <stdio.h>
    void main(void)
        {
         char far *ptr;
         unsigned seg,off;

         ptr=MK_FP(0x8000,0);
         seg=FP_SEG(ptr);
```

```
    off=FP_OFF(ptr);
    printf("farptr=%Fp, segment=%04x, offset=%04x\n",
                                        ptr,seg,off);
   }
```

```
int peek(unsigned segment,unsigned offset);
char peekb(unsigned segment,unsigned offset);
void poke(unsigned segment,unsigned offset,int value);
void pokeb(unsigned segment,unsigned offset,char value);
```

Cele patru funcții de mai sus realizează citirea (peek, peekb) respectiv scrierea (poke, pokeb) unui cuvânt sau caracter de la sau la adresa de memorie segment:offset.

12.2. Funcții pentru accesul la porturile de intrare sau ieşire.

```
int inp(int portid);
int outp(int portid, int value);
int inport(int portid);
unsigned char inportb(int portid);
void outport(int portid,int value);
void outportb(int portid,unsigned char value);
```

Toate cele şase funcții de mai sus au drept scop citirea (inport) respectiv scrierea (outport) a unui cuvânt, sau caracter (inp, inportb, outp, outportb) de la, sau la portul de intrare sau ieşire specificat prin variabila portid.

12.3. Funcții pentru gestiunea intervalelor de timp

```
void delay(unsigned miliseconds);
```

Funcția suspendă execuția programului în intervalul specificat (în milisecunde).

```
void sleep(unsigned seconds);
```

Funcția suspendă execuția programului curent pe durata seconds secunde.

12.4. Functii pentru controlul sistemului de întreruperi

```
void disable(void);
```

Această funcție invalidează întreruperile, lăsând validă numai întreruperea nemascabilă NMI. Este echivalentă cu instrucțiunea CLI din limbajul de asamblare.

```
void enable(void);
```

Această funcție validează întreruperile oferind programatorului controlul flexibil al întreruperilor hardware. Este echivalentă cu instrucțiunea STI din limbajul de asamblare.

Introducere în programare

12.5. Funcții pentru alocarea memoriei (prin DOS)

*int allocmem(unsigned size, unsigned *segp);*

Funcția apelează DOS 0x48 pentru a aloca o zonă de memorie liberă de dimensiune *size**16 octeți. Ea returnează adresa de segment a blocului alocat în locația pointată de *segp*. În caz de succes returnează -1. În caz contrar se setează _doserrno si variabila globală errno.

int setblock(unsigned segx, unsigned newsize);

Acestă funcție modifică dimensiunea alocată unui bloc de memorie, de la adresa de segment *segx* rămânând alocați doar *newsize**16 octeți. În caz de succes funcția returnează -1. Funcția face apelul DOS 0x4A.

int freemem(unsigned segx);
Funcția eliberează un bloc de memorie alocat cu allocmem. Parametrul segx este adresa de segment returnată de aceasta din urmă. În caz de succes funcția returnează -1. Funcția face apelul DOS 0x44.

12.6. Întreruperi software. Accesul la registrele procesorului.

void geninterrupt(int intr_num);

Acest macrou generează întreruperea software intr_num. Starea registrelor după apel depinde de întreruperea apelată. Accesul la registrele procesorului se face prin variabilele pseudoregistru prezentate mai jos:

_AX, _BX, _CX, _DX, _SI, _DI, _BP, _SP, _DS, _CS, _ES, _SS, _FLAGS

care le copie starea. Aceste variabile trebuie folosite cu precauție. Se recomandă a se evita alterarea variabilelor _DS, _SS, _CS si _SP. În orice caz, se va urmări codul generat şi se va depana cu atenție programul.

*int int86(int intno, union REGS *inregs, union REGS *outregs);*

Aceasta funcție execută întreruperea software specificată în intno. Înaintea execuției întreruperii, se copie în registre valorile specificate în reuniunea pointată de inregs. După execuție, valorile registrelor sunt copiate în outregs; de asemenea, şi starea bitului Carry în x.cflag (1=de obicei indică o eroare). De notat că outregs poate pointa pe aceeaşi structura ca inregs. Funcția returnează valoarea din AX la terminarea întreruperii. Reuniunea REGS este definită astfel:

```
struct WORDREGS {
    unsigned int ax,bx,cx,dx,si,di,cflag,flags;};
struct BYTEREGS {
    unsigned char al,ah,bl,bh,cl,ch,dl,dh;};


union REGS {
    struct WORDREGS x;
    struct BYTEREGS h; };
```

*int int86x(int intno, union REGS *inregs, union REGS *outregs, struct SREGS *segregs);*

Aceasta funcție este identică ca funcțiune cu cea anterioară, permițând în plus folosirea registrelor de segment pentru transmiterea de parametri (salvează si restaurează automat ES si DS). Returnează aceleaşi valori ca int86. Structura SREGS este definită astfel:

```
struct SREGS {
    unsigned int es;
    unsigned int cs;
    unsigned int ss;
    unsigned int ds;};
```

*void intr(int intno, struct REGPACK *preg);*

Funcția realizează o alternativă pentru interfațarea cu întreruperile software, numărul întreruperii fiind specificat în variabila intno. Structura REGPACK este definită astfel:

```
struct REGPACK {
    unsigned r_ax,r_bx,r_cx,r_dx;
    unsigned r_bp,r_si,r_di,r_ds,r_es,r_flag;};
```

*void segread(struct SREGS *segp);*

Funcția returnează în structura pointata de segp valorile registrelor de segment.

12.7. Funcții pentru citirea sau modificarea vectorilor de întrerupere

*void interrupt (*getvect(int interruptno))();*

Fiecare procesor din familia 80x86 include un set de vectori de întrerupere, numerotați de la 0 la 255. Valoarea pe 4 octeți din fiecare vector este o adresă reprezentând adresa rutinei de tratarea a întreruperii. getvect citeşte valoarea vectorului de întrerupere specificat de interruptno şi returnează această valoare ca un pointer far la o rutină de tratare a întreruperii.

*void setvect(int interruptno, void interrupt (*isr)());*

Această funcție setează valoarea vectorului de întrerupere specificat prin interruptno cu o valoare nouă, isr, care este un pointer far la adresa altei funcții de tratare a întreruperii. Obligatoriu aceasta nouă funcție trebuie sa fie declarată de tip interrupt (cu modificatorul interrupt). Exemplu:

```
#include <stdio.h>
#include <dos.h>

void interrupt (*oldvect)();
void interrupt get_out();
void capture_prtscr(void interrupt (*func)());

int looping=1;

void main(void)
    {
     puts("Pentru a termina apasati <Shift><PrtSc>");
     capture_prtscr(get_out);
     while(looping);
     puts("Succes");
    }

/*noua rutina de tratare intrerupere*/
 void interrupt get_out()
    {
     setvect(5,oldfunc);
     looping=0;
    }

void capture_prtscr(void interrupt (*func)())
    {
       oldfunc=getvect(5);
       setvect(5,func);
    }
```

12.8. Apeluri de funcții DOS.

int bdos(int dosfun, unsigned dosdx, unsigned dosal);

Furnizează accesul direct la multe funcții DOS. Ca argumente:

dosfun = numărul funcției DOS;
dosdx = valoarea registrului DX;
dosal = valoarea registrului AL.

Funcția returnează valoarea registrului AX. Exemplu:

```
#include <stdio.h>
#include <dos.h>

char current_drive(void);

main()
   {
    printf("DRIVERUL %c\n",current_drive());
    getch();
   }

char current_drive(void)
   {
    char drive;
    drive = bdos(0x19,0,0);
    return ('A'+drive);
   }
```

*int bdosptr(int dosfun, void *argument, unsigned dosal);*

Funcția furnizează accesul la funcțiile DOS prin argumentele sale:

dosfun = numărul funcției DOS;
argument = pointer cerut de funcție in registrul DS:DX;
dosal = valoarea registrului AL.

Returnează aceleaşi valori ca bdos.

*int intdos(union REGS *inregs, union REGS *outregs);*

Funcția apelează întreruperea INT 21h pentru a realiza funcția sistem specifică în registrul AH (inregs->h.ah). Structura outregs este actualizată setându-se in caz de eroare indicatorul carry.

*int intdosx(union REGS *inregs, union REGS *outregs,*
*struct SREGS *segregs);*

Este similară funcției intdos, permițând în plus şi transmiterea parametrilor prin registrele de segment (DS,ES). Aceste registre sunt restaurate la terminarea apelului. Exemplu:

```
#include <stdio.h>
#include <dos.h>
```

```
int delete_file(char far *filename);

main()
    {
     int err;
     err=delete_file("Temp1.$$$");
     printf("S-a sters fisierul Temp1.$$$:%s\n",(!err)?"DA":"NU");
    }

int delete_file(char far *filename)
    {
    union REGS regs;
    struct SREGS sregs;
    int ret;

    regs.h.ah=0x41;     /*delete file*/
    regs.x.dx=FP_OFF(filename);
    sregs.ds=FP_SEG(filename);
    ret=intdos(&regs,&regs,&sregs);
    return(regs.x.cflag?ret:0);
    }
```

12.9. Funcții pentru terminarea sau lansarea programelor

void exec(int code);

Funcția determină terminarea normală a programului şi furnizarea unui cod de retur (code) la ieşire. Acest cod poate fi interpretat ulterior în fişierele de comenzi. Este echivalentă cu funcția DOS 0x4C.

void abort(void);

Provoacă terminarea anormală a programului, cu afişarea unui mesaj corespunzător pe consola sistemului.

void keep(unsigned char status,unsigned size);

Apelul acestei funcții permite terminarea programului curent cu transmiterea stării procesului părinte în variabila status, precum şi setarea lui ca program TSR (Terminate and Stay Resident) de dimensiune size paragrafe. Restul memoriei este eliberat. Funcția este similară apelului DOS 0x31.

*int system(const char *cmd);*

Lansează COMMAND.COM şi execută o comandă internă, sau o comandă externă, sau un fişier de comenzi. Şirul de caractere, folosit ca parametru, conține numele complet al comenzii şi toate argumentele necesare execuției.

unsigned getpsp(void);

Funcția returnează segmentul prefix (Program Segment Prefix) al programului curent. Ea realizează functia DOS 0x62. În locul acestei funcții însă se poate folosi direct valoarea variabilei globale _psp.

int bioskey(int cmd);

Pot fi executate serviciile specifice INT 16h. Cu ajutorul acestei funcții se pot citi codurile de scanare, codurile extinse ale tastelor, se pot citi caractere fără extragere din bufferul tastaturii, sau se poate afla starea tastelor speciale (SHIFT, CTRL, ALT):

- cmd=0 citeşte codul de scanare si scoate caracterul din bufferul tastaturii; dacă nu mai sunt caractere se aşteaptă apăsarea unei noi taste.
- cmd=1 citeşte codul de scanare fără a scoate caracterul din bufferul tastaturii; dacă nu mai sunt caractere funcția returnează 0.
- cmd=2 citeşte "BIOS Shift State" (starea tastelor speciale SHIFT, CTRL, ALT).

int biosprint(int cmd, int abyte, int lpt_port);

Reuneşte toate funcțiile specifice INT 17h (accesul la portul paralel LPTxx). Funcția permite citirea stării, trimiterea unui caracter (abyte), sau inițializarea imprimantei:
- cmd=0 trimite caracterul abyte la portul paralel cu numărul dat de lpt_port (0=LPT1, 1=LPT2, etc).
- cmd=1 inițializează imprimanta, conectată pe lpt_port, cu valorile din abyte.
- cmd=2 citeşte starea imprimantei conectate pe lpt_port.

int bioscom(int cmd, int abyte, int com_port);

Reuneşte toate funcțiile specifice INT 14h (accesul la portul serial COMxx). Funcția permite citirea stării, trimiterea unui caracter (abyte) pe comunicație, citirea unui caracter de pe comunicație, sau setarea unor controale şi parametri de funcționare:

- cmd=0 inițializarea portului de comunicație com_port (0=COM1, 1=COM2) cu parametrii dați de abyte.
- cmd=1 trimite caracterul abyte pe linia de comunicație data de com_port.
- cmd=2 citeşte caracterul recepționat pe linia de comunicație dată de com_port.
- cmd=3 citeste starea portului de comunicație com_port.

long biostime(int cmd, long newtime);

Funcția permite citirea (cmd=0), sau modificarea (cmd=1) timpului sistem, exprimat în incrementi de 55ms, de la pornirea sistemului.

12.10 Dezvoltarea de cod într-un limbaj mixt (C şi ASM)

Este specifică mediului de dezvoltare Borland C care oferă posibilitatea inserării de cod ASM în programe sursă C. Fiecare instrucțiune ASM trebuie precedată de cuvântul cheie asm sau trebuie inclusă într-un bloc de tip ASM, de forma:

```
asm {
; instrucțiuni ASM
    }
```

Acest model de dezvoltare este foarte comod, deoarece vizibilitatea obiectelor definite în C se extinde şi asupra blocurilor ASM. Astfel putem folosi nume de parametri formali ai funcțiilor etc., fără a mai scrie caracterul "_" în fața numelor respective. Se evită astfel secvențele de intrare şi revenire din funcții, care sunt realizate automat de compilatorul C.

Există restricții de utilizare a etichetelor. Mai precis, etichetele trebuie definite ca etichete C, deci în afara blocurilor ASM.

Ca şi la dezvoltarea pe module separate, programatorul trebuie să gestioneze explicit modelele de memorie, în sesnsul definirii precise a adreselor far sau near. Pentru aceasta, se utilizează modelele de memorie C şi/sau cuvintele cheie near şi far.

Pentru exemplificare să discutăm implementarea unei funcții de căutare folosind atât metoda clasică cât şi metoda cu cod de asamblare.

Funcția respectivă scrisă în C este prezentată mai jos:

```
int my_search(int *x, size_t n, int y)
    {
    int i;
    for(i=0;i<n;i++)
        if(y==x[i])return i;
        return –1; // nu am găsit întregul y  în tabloul x
    }
```

Funcția rescrisă folosind limbaj de asamblare :

```
int my_search(int near *x, size_t n, int y)
    {
        asm {
        push si
        push cx
            push dx
            mov si,x
            mov cx,n
            jcxz nu
            mov dx,x
            sub si,2
            }
```

```
et:  asm {
      add si,2
      cmp [si], dx
      loopnz et
      jnz nu
          mov ax,n
          sub ax,cx
          dec ax
          jmp m_end
         }

}

nu:
      asm mov ax,-1
m_end:
     asm {
      pop dx
      pop cx
      pop si
      }
return _AX
}
```

Se observă că indicele găsit a fost calculat ca diferență dintre dimensiunea n a tabloului şi valoarea incrementată a contorului Cx la ieşirea din buclă. Revenirea din buclă se face folosind valoarea stocată în registrul AX la care am acces prin preudoregistrul _AX.

Metoda are dezavantajul scăderii portabilității la recompilarea sub alte compilatoare.

12.11 Tratarea dispozitivelor bus asincrone

Să presupunem că se dezvolta un cod pentru controlul unui dispozitiv hardware (un driver de exemplu). Acest lucru este realizat prin depunerea de diverse valori în registre hard la adrese absolute (existente fizic independent de modul virtual de manipulare a memoriei pus la dispoziție de SO) .

Să presupunem că acest dispozitiv are 2 registre, fiecare fiecare având lungimea de 16 biți la adrese de memorie crescătoare.

Metoda clasică de accesare este ca mai jos :

```
struct devregs
        {
        unsigned short csr   // control si stare
        unsigned short data  // portul de date
        }
```

```
#define  ER  0x1
#define  RDY  0x2
#define  RES  0x4

// adresa absoluta a dispozitivului

#define DEVADR  (( struct dev regs*)ox  ffff0000

// numărul de dispozitive de acest tip din sistem

#define NDEVS 4
```

// o funcție ce citeşte un octet din dispozitivul n folosind protocol de tip busy/wait.

Aşteaptă pâna când apare semnal de RDY (ready) sau ER(error); dacă nu este eroare şi un ready activ înseamnă că pot citi un octet, altfel va anula eroarea dar voi întoarce 0xffff.

```
unsigned int read_dev ( unsigned devno)
  {
   unsigned  devregs *dvp=DEVADR + devno;
   if ( devno >= NDEVS ) return (0xffff)
                    // nu există un asemenea dispozitiv în lot
   while ((  dvp -> csr& (RDY | ER )) == 0);
                    // bucla vector care aşteaptă prima una din condiții este validată
   if  dvp -> csr & ER    // a fot eroare
           {
           dvp -> csr= RES   // reset
           return 0xffff;
           }
return (dvp-> data) & 0xffff;   // nu  a fost eroare şi dispozitivul e gata să emită octetul
  }
```

13. Preprocesorul

Orice program C, în faza de compilare trece întâi printr-o preprocesare şi abia apoi se va genera codul echivalent în asamblare. Preprocesorul după cum îi spune şi numele, face prima procesare asupra sursei C conform directivelor incluse direct sau indirect, prin includere de bilioteci, de către programator.

Directivele preprocesorului sunt linii sursă al căror prim caracter diferit de spațiile albe este # (diez). Liniile care conțin directive ale preprocesorului sunt eliminate din textul sursă înainte ca acesta să fie citit de compilator. Preprocesorul înlocuieşte orice secvență de spații albe (inclusiv comentarii) cu un singur spațiu.

13.1. Directiva nulă

O linie care conține numai caracterul # (în afară de eventualele spații albe) este ignorată de preprocesor.

13.2. Directivele #define şi #undef

"#" "define" identificator [["(" lista_argumente ")"] corp] .
"#" "undef" identificator

Directiva #define defineşte o macroinstrucțiune. Dacă macroinstrucțiunea are argumente, paranteza stângă trebuie să apară în definiție lipită de numele definit. Argumentele se separă prin virgulă. Corpul macrodefiniției nu poate avea mai mult de o linie sursă (se poate continua linia cu \).

În faza de preprocesare, apariția numelui de macro în textul sursă (eventual urmat de o listă de argumente actuale in paranteze) este înlocuită cu corpul de macro, în care parametrii formali au fost înlocuiți cu textul parametrilor actuali.

După substituție, textul obținut este citit din nou, până când nu mai sunt necesare substituții (directivele preprocesorului nu sunt recunoscute în textul produs prin expandarea unei macroinstrucțiuni). Numele de macro nu sunt recunoscute în şiruri de caractere sau comentarii. O macroinstrucțiune nu se poate apela recursiv.

La apelul unei macro cu argumente, numai virgulele de pe primul nivel sunt luate ca separatoare de argumente (nu şi cele care apar în paranteze sau între apostrofe sau ghilimele).

În corpul unui macro se pot folosi următorii operatori care nu acționează decât în faza de preprocesare:

- ## care dispare total, nerămânînd nici măcar un spațiu, şi
- # care, plasat in fața unui argument formal, îl înlocuieşte cu argumentul actual PLASAT ÎN GHILIMELE.

Directiva #undef şterge o macrodefiniție din tabela de macrodefiniții.

13.3. Directiva #include

#include <nume_fisier>
#include "nume_fisier"
#include nume_macro

Numele de macro până la urmă trebuie să se expandeze în ceva de forma <nume_fisier> sau "nume_fisier". Efectul directivei constă din inserarea în unitatea de compilare, în locul directivei #include, a textului din fişierul respectiv. Textul inserat poate conține şi el directive de preprocesare, inclusiv directiva #include.

Dacă în numele de fişier apare şi o cale de acces, se va căuta numai acolo. Altfel, dacă s-a folosit forma cu ghilimele, se caută în directorul curent. Urmează, dacă tot nu s-a localizat fişierul, să se caute pe rând în directoarele din lista dată la instalare, sau pe linia de comandă, sau în mediul integrat.

13.4. Directive de compilare condiționată

"#" "if" expresie_constantă_1
[text_1]
["#" "elif" expresie_constantă_2
text_2]...
["#" "else"
text_else]
"#" "endif"

Dacă expresia_1 nu este nulă, atunci se compilează text_1; altfel, dacă expresia_2 nu este nulă, atunci se compilează text_2; etc.; altfel, dacă este prezentă directiva #else, se compilează text_else. Directivele de compilare condiționată pot fi îmbricate.

Expresia condițională poate conține operatorul de preprocesare defined. Acesta are un argument, un nume de macro eventual în paranteze. Valoarea expresiei defined (nume) este 1 daca numele este definit cu #defined, sau 0 dacă nu există o macro cu acel nume. Un identificator definit cu un corp de macro vid se consideră definit.

#ifdef nume ... echivalentă cu #if defined (nume)
#ifndef nume ... echivalentă cu #if !defined (nume)

13.5. Directiva #line

#line număr_linie ["nume_fisier"]

Efectul directivei constă în aceea că în fişierul produs de C (preprocesorul C oferit ca utilitar) sau in mesajele de eroare, etc. liniile care urmează după #line vor fi referite ca şi cum ar începe cu linia numărul număr_linie şi ar veni din fişierul nume_fişier.

De obicei directiva #line nu apare în programele scrise cu mâna, ci mai ales în programele generate automat. Dacă numele de fişier lipseşte el rămâne cel valabil în acel moment.

În argumentele directivei #line, macroinstrucțiunile sunt expandate, ca şi la directiva #include.

13.6. Directiva #error

#error mesaj

Efectul ei constă în generarea unui mesaj de eroare de forma

Error: nume_fişier line număr_linie : Error directive : mesaj

Compilarea unei directive #error opreşte compilarea. De obicei apare în fişierele antet (incluse cu directiva #include), în cadrul unei secvențe compilate condiționat. De exemplu:

```
#if !defined ( __cplusplus )
    #error Nu se poate compila decit in C++
#endif
```

13.7. Directivele #pragma

#pragma argsused

Poate apare numai în afara unei funcții. Are ca efect suprimarea mesajului de avertisment referitor la neutilizarea unui parametru formal, dar numai în prima funcție compilată după apariția directivei.

#pragma exit nume_functie [prioritate]
#pragma startup nume_functie [prioritate]

Inserează funcțiile citate (definite ca void nume_functie (void)) pentru apel înainte de main (respectiv după terminarea funcției main). Prioritatea este un întreg de la 0 la 255. Funcțiile cu priorități mai mici vor fi apelate mai devreme la lansare şi mai târziu la terminarea execuției. Prioritățile de la 0 la 63 sunt rezervate pentru rutinele din bibliotecile Turbo. Dacă prioritatea lipseşte, se ia 100. Funcțiile citate trebuie declarate înaintea apariției directivei.

#pragma inline

Anunță compilatorul că în program există cod inline în limbaj de asamblare. Este echivalentă cu opțiunea de compilare -B. Întâlnirea directivei #pragma inline, dacă opțiunea -B nu este activă, are ca efect relansarea compilării cu opțiunea -B activată.

#pragma option

#pragma option optiune [,optiune]...

Opțiuni de compilare care pot apare în #pragma option NUMAI ÎNAINTE DE FOLOSIREA UNUI NUME DE MACRO CARE ÎNCEPE CU DOUĂ LINIUȚE DE SUBLINIERE, SAU ÎNAINTE DE APARIȚIA PRIMULUI ATOM LEXICAL C:

-E nume_asamblor
-f opțiune_flotantă
-i număr_caractere_semnificative
-m model_memorie
-n director_ieşire
-o nume_fisier_ieşire
-u (generare de liniuțe de subliniere pentru globali)
-z nume_segment

Opțiuni care pot apare numai dacă apar în afara oricăror declarații de obiecte sau funcții:

-1 (instrucțiuni 80186)
-2 (instrucțiuni 80286)
-a (controlul alinierii)
-ff (control "rapid" al operațiilor în virgula mobilă)
-G (optimizare pentru viteză)
-k (control normal al stivei)
-N (controlul stivei)
-O (controlul optimizării)
-p (secvență de apel Pascal)
-r (alocare registre)
-v (control pentru depanare)
-y (control asupra numerelor de linii în codul obiect)

Opțiuni care pot fi date oriunde si au efect imediat:

-A (setul de cuvinte cheie)
-C (comentarii îmbricate)
-d (contopirea şirurilor duplicate)
-g număr (oprire după număr avertismente)
-j număr (oprire după număr erori)
-K (tipul char este fără semn)

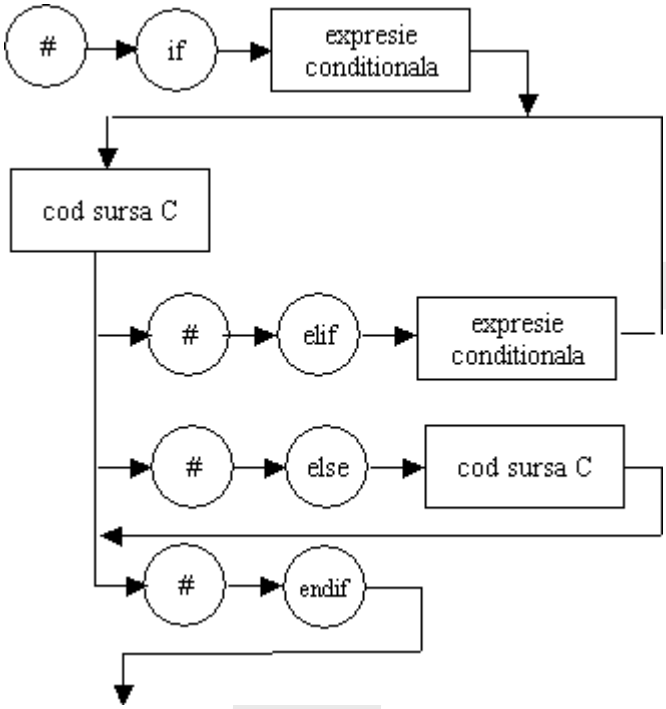
#pragma warn
#pragma warn +xxx Avertismentele xxx sunt activate
#pragma warn -xxx Avertismentele xxx sunt inactivate

După cum am văzut preprocesorul permite şi declararea anumitor zone de cod astfel încât ele sa fie compilate în anumite condiții cum ar fi :

- ➔ nu sunt definite anumite constante sau variabile locale
- ➔ codul este util, dar pentru depanare

➔ doresc să impiedic folosirea unor funcții din bibliotecă întru-cât sunt rescrise de oricine(supraîncărcarea funcțiilor)

Sintaxa generala :



Exemplu 1.
pentru headere personale să zicem fişierul test.h

#ifndef __TEST
#define __TEST
.....
corpul
.....
#endif

Sau pentru depanare
#if DEBUG
printf(".....");
#endif

Sau daca nu am definite ni;te valori speciale
#ifndef M_PI
#define M_PI 3.1415....
#endif.

Scurtă prezentare a regulilor de bază de care trebuie ţinut cont la proiectarea unei applicaţii

Introducere în programare

Stil de programare nerecomandat	Stil recomandat
Punerea declaraţiilor cu extern în fişierul.c	Referirile la funcţie/variabilă extern Se pun în fişierele header.h
Accesul comun la date între funcţii realizat prin variabile globale	Accesul comun se face prin transfer de argument
Folosirea de date globale când nu e cazul	Se rezolvă prin declarare de tip static
Crearea a numeroase funcţii cu scopuri speciale	Creearea unor primitive generale
Utilizarea numelor de fişier sau variabilă fără legătură cu contextul	Folosirea de nume care se apropie de pseudocod
Folosirea constante numerice	Se definesc nişte nume pt. acele valori
Folosirea goto	Folosirea structurilor de control a fluxurilor
Scrirea de cod redundant	Se folosesc funcţii în cazul în care se observă acelaşi lucru

```
// calculul randamentului si puterii pentru un fir conectat la un generator de cc
#include <stdio.h>
#include <conio.h>
#include <math.h>

void main(void)
{
// incepe zona declarare variabile

int e,ri,l,s,imax;
float r,pext,eta,i,g;
//sfirsit  zona declarare variabile

// incepe zona citire date de intrare
clrscr();
printf("\nTensiunea gen. E=");
scanf("%d",&e);
printf("\nRez. interna gen. ri=");
scanf("%d",&ri);
printf("\nLungimea fir L=");
scanf("%d",&l);
printf("\nSectiunea fir S=");
scanf("%d",&s);
printf("\nCurentul maxim admis Imax=");
scanf("%d",&imax);
printf("\nConductivitatea gama=");
scanf("%f",&g);
// sfirsit  zona citire date de intrare

//incepe zona de calcul conform algoritmului
        r=l/(g*s);
        i=e/(r+ri);
        pext=r*sqrt(i);
        eta=pext/(e*i);
// sfirsit zona de calcul conform algoritmului

//incepe zona afisarii rezultatelor
printf("\nPext=%3.8f",pext);
printf("\nRandament=%3.8f",eta);
if(i<imax) printf("\nI=%3.8f < Imax",i);
else printf("\nI=%2.8f >Imax",i);
//sfirsit zona afisarii rezultatelor

printf("\nApasati orice tasta ca sa terminati programul");
getch();
}
```

```
/*************************************************************************
*************************************************************************
***************************/
//exemplu de calcul a unei functii cu mai multe ramuri;
#include <stdio.h>
#include <conio.h>

void main(void)
{
int x,f;

clrscr();
printf("x=");
scanf("%d",&x);

if(x<-5)f=x-1;
else
        if(x==-5)f=-6;
        else f=2*x+4;

printf("\nf(x)=%d",f);
getch();
}
/*************************************************************************
*************************************************************************
***************************/
// repetarea de un numar specificat de ori a calcului
// pentr o functie_multipla;
#include <stdio.h>
#include <conio.h>

void main(void)
{
int x,n,f,i;

begin
clrscr;
printf("Numarul de repetari a calculului functiei f n=");
scanf("%d",&n);

for(i:=0;i<n;i++)
{
clrscr();
printf("x=");
scanf("%d",&x);
```

```
if(x<-5)f=x-1;
else
        if(x==-5)f=-6;
        else f=2*x+4;

printf("\nf(x)=%d",f);
}
getch();
}

/**********************************************************************
**********************************************************************
***************************/
// exemplu de lucru cu sirurile de caractere
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
{
char s[10],a,c;
int i,b,d;

clrscr();
printf("Introduceti sirul de prelucrat s=");
gets(s);

a=s[0];
b=strlen(s);
c=s[b-1];

if ((a>='A')&&(a <='Z'))
        {
        s[0]+=32;
        s[b-1]+=32;
        }
else
        {
        s[0]-=32;
        s[b-1]-=32;
        }
printf("\nprima litera %c",a);
printf("\nultima litera %c",c);
printf("\nsirul prelucrat %s",s);
getch();
```

```
//obs programul nu ia in calcul situatia in care in sir de litere apar si cifre
}
/*************************************************************************
*************************************************************************
***************************/
// program care implementeaza functia de stergere a unui caracter
#include <stdio.h>
#include <conio.h>
#include <string.h>

int strdel(char s[], int pos);
void main(void)
{
char s[20];
int n;
clrscr();
printf("Dati sirul s=");
gets(s);
printf("dati pozitia de eliminare ");
scanf("%d",&n);
if(strdel(s,n))puts("Eroare, pozitia este in afara sirului ");
else printf("\n Noul sir este=%s",s);
getch();
}
int strdel(char s[], int pos)
{
if(pos>=strlen(s))return -1;// eroare de apelare
pos--;
while(s[pos]!='\0')
        {
        s[pos]=s[pos+1];
        pos++;
        }
return 0;// functia s-a terminat corect
}
/*************************************************************************
*************************************************************************
***************************/
// program care implementeaza functia de stergere a unui caracter
// existenta in pascal
#include <stdio.h>
#include <conio.h>
#include <string.h>

int strdelpas(char s[], int pos, int nrcars);
void main(void)
```

```
{
char s[20];
int n,nrc;
clrscr();
printf("Dati sirul s=");
gets(s);
printf("dati pozitia de eliminare ");
scanf("%d",&n);
printf("dati nr de caractere ce se vor sterge ");
scanf("%d",&nrc);

if(strdelpas(s,n,nrc))puts("Eroare, pozitia este in afara sirului");
else printf("\n Noul sir este=%s",s);
getch();
}
int strdelpas(char s[], int pos, int nrcars)
{
if(pos>=strlen(s))return -1;// eroare de apelare
pos--;
while(s[pos]!='\0')
        {
        s[pos]=s[pos+nrcars];
        pos++;
        }
return 0;// functia s-a terminat corect
}
/*************************************************************************
*************************************************************************
***************************/
//program pentru eliminarea caracterelor indentice;
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(void)
{
char s[25];
int i=0,j,n,pos;

clrscr();
printf("Introduceti sirul de prelucrat s=");
scanf("%s",s);

n=strlen(s);

do
{
```

```
        j=i+1;
                do
                {
                        if(s[i]==s[j])
                                {
                        pos=j;
                        while(s[pos]!='\0')
                                {
                                s[pos]=s[pos+1];
                                pos++;
                                }


                                n--;
                                }
                        else j++;
                }while (j<n);
        i++;
}while (i<n);

printf("Sirul rezultat este s=%s",s);
getch();
}
/*****************************************************************************
*****************************************************************************
***************************/
//program pentru eliminarea caracterelor indentice folosind functii;
// transfer de parametri in vechea maniera de programare
#include <stdio.h>
#include <conio.h>
#include <string.h>

int strdel(char s[], int pos);
void main(void)
{
char s[25];
int i=0,j,n,pos;

clrscr();
printf("Introduceti sirul de prelucrat s=");
scanf("%s",s);

n=strlen(s);

do
{
```

```
        j=i+1;
                do
                {
                        if(s[i]==s[j])
                                {
                                strdel(s,j);
                                n--;
                                }
                        else j++;
                }while (j<n);
        i++;
}while (i<n);

printf("Sirul rezultat este s=%s",s);
getch();
}
int strdel(char s[], int pos)
{
if(pos>=strlen(s))return -1;// eroare de apelare
while(s[pos]!='\0')
        {
        s[pos]=s[pos+1];
        pos++;
        }
return 0;// functia s-a terminat corect
}
/*************************************************************************
*************************************************************************
***************************/

// program pentru ordonarea unui vector
// algoritm FOARTE lent pentru valori mari ale lui n
#include <stdio.h>
#include <conio.h>

void main(void)
{
float v[100],temp;
int i,n,j;

clrscr();

printf("Introduceti dimensiunea vectorului n=");
scanf("%d",&n);

for(i=0;i<n;i++)
```

```
        {
         printf("V[%d]=",i);
         scanf("%f",&v[i]);
        }
for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
                if(v[i]>v[j])
                {
                 temp=v[i];
                 v[i]=v[j];
                 v[j]=temp;
                 };
puts("Vectorul ordonat");
for(i=0;i<n;i++)
        printf("%3.3f ",v[i]);
}
/*************************************************************************
*************************************************************************
***************************/

//program pentru extragerea unui vector ce contine
//elementele mai mari ca 1 dintr-un matrice;
// se foloseste validarea input-ului
#include <stdio.h>
#include <conio.h>

void main(void)
{
int n,m,i,j,k;
float a[10][10], b[100];

clrscr();
printf("n=");
scanf("%d",&n);
printf("m=");
scanf("%d",&m);

while ((n<0)||(n>10)||(m<0)||(m>10))
        {
        puts("0<m,n<10");
        printf("n=");
        scanf("%d",&n);
        printf("m=");
        scanf("%d",&m);
        }
for(i=0;i<n;i++)
```

```
        for(j=0;j<m;j++)
                {
                printf("m[%d,%d]=",i,j);
                scanf("%f",&a[i][j]);
                };
k=0;
for(i=0;i<n;i++)
        for(j=0;j<m;j++)
                if(a[i][j]>1)
                        b[k++]=a[i][j];
if(!k)puts("\nMatricea nu are elemente > 1");
else
        {
        puts("vectorul elem. >1 este:");
        for(i=0;i<k;i++)printf("%3.2f ",b[i]);
        };
getch();
}

/*************************************************************************
*************************************************************************
***************************/
//program inmutire matrici Amn x Bpq;
#include <stdio.h>
#include <conio.h>

void main(void)
{
int m,n,p,q,i,j,k;
float a[10][10], b[10][10],c[10][10];

clrscr();
        printf("m=");
        scanf("%d",&m);
        printf("n=");
        scanf("%d",&n);
        printf("p=");
        scanf("%d",&p);
        printf("q=");
        scanf("%d",&q);

while (n!=p)
        {
        puts("Date eronate, n trebuie sa fie egal cu p");
        printf("m=");
        scanf("%d",&m);
```

```
        printf("n=");
        scanf("%d",&n);
        printf("p=");
        scanf("%d",&p);
        printf("q=");
        scanf("%d",&q);

    }
for(i=0;i<m;i++)
        for(j=0;j<n;j++)
                {
                printf("a[%d,%d]=",i,j);
                scanf("%f",&a[i][j]);
                };
for(i=0;i<p;i++)
        for(j=0;j<q;j++)
                {
                printf("b[%d,%d]=",i,j);
                scanf("%f",&b[i][j]);
                };

for(i=0;i<m;i++)
        for(j=0;j<q;j++)
                {
                c[i][j]=0;
                for(k=0;k<n;k++)
                        c[i][j]=c[i][j]+a[i][k]*b[k][j];
                };
clrscr();
puts("Matricea C=A*B este ");
for(i=0;i<m;i++)
    {
        for(j=0;j<q;j++)
                printf("%3.2f ",c[i][j]);
        puts("");
        }
getch();
}

/*****************************************************************************
*****************************************************************************
***************************/
// program care generaza vectorul ce contine
// suma elem de pe liniile unei matrici;
#include <stdio.h>
#include <conio.h>
```

```
void main(void)
{
int n,m,i,j,k;
float a[10][10], b[10],s;

clrscr();
printf("n=");
scanf("%d",&n);
printf("m=");
scanf("%d",&m);

for(i=0;i<n;i++)
        for(j=0;j<m;j++)
                {
                printf("m[%d,%d]=",i,j);
                scanf("%f",&a[i][j]);
                };


for(j=0;j<m;j++)
      {
        s=0;
        for(i=0;i<n;i++)
                s+=a[i][j];
                b[j]=s;
        }
for(i=0;i<n;i++)
      {
        for(j=0;j<m;j++)
                printf("%3.2f ",a[i][j]);
        puts("");
        }

puts("Vectorul continind suma elementelor pe fiecare linie este");
for(i=0;i<m;i++)
        printf("%3.2f ",b[i]);
getch();
}
/*****************************************************************************
*****************************************************************************
***************************/
// program exemplu de functie;
#include <stdio.h>
#include <conio.h>
#include <math.h>
```

```
#include <values.h>

float rezolv(float x, float y);

void main(void)
{
float a,b,x;
clrscr();
printf("a=");
scanf("%f",&a);
printf("b=");
scanf("%f",&b);
x=rezolv(a,b);
printf("\nSolutia este %8.3f",x);
getch();
}

float rezolv(float x, float y)
{
float fx,fy,fc,c,eps;
fx=x*sqrt(x)+5*sqrt(x)+2*x-7;
fy=y*sqrt(y)+5*sqrt(y)+2*y-7;
if(fx*fy<0) return MAXDOUBLE; // semnalez o eroare de calcul
do
{
        c=(x+y)/2;
        fc=c*sqrt(c)+5*sqrt(c)+2*c-7;
        if (fx*fc>0)
                {
                fx=fc;
                x=c;
                }
        else y=c;
        eps=abs(x-y);
}while(eps>0.001);
return c;
}

/*
program care realizeaza sumarea a doi vectori de aceeasi
dimensiune, versiune cu for

*/
#include <stdio.h>
#include <conio.h>
```

```
void main(void)
{
int i,n;
int x[20],y[20],z[20];

clrscr();
printf("n=");
scanf("%d",&n);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }

for(i=0;i<n;i++)
        {
        printf("y[%d]=",i);
        scanf("%d",&y[i]);
        }
for(i=0;i<n;i++)
        {
        z[i]=x[i]+y[i];
        }
for(i=0;i<n;i++)
        {
        printf("\nz[%d]=%d",i,z[i]);
        }
getch();
}
/*************************************************************************
*************************************************************************
***************************/

/*
program care realizeaza sumarea a doi vectori de aceeasi
dimensiune, versiune cu while

*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
int i,n;
int x[20],y[20],z[20];
```

```
clrscr();
printf("n=");
scanf("%d",&n);

i=0;
while(i<n)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        i++;
        }
i=0;
while(i<n)
        {
        printf("y[%d]=",i);
        scanf("%d",&y[i]);
        i++;
        }
i=0;
while(i<n)
        {
        z[i]=x[i]+y[i];
        i++;
        }
i=0;
while(i<n)
        {
        printf("\nz[%d]=%d",i,z[i]);
        i++;
        }
getch();
}
/*************************************************************************
*************************************************************************
***************************/
/*
program care realizeaza sumarea a doi vectori de aceeasi
dimensiune, versiune cu do..while
Observatie nu este o echivalnta perfecta a for.. pentru ca testul fiind
posterior bucla se executa macar o data !

*/
#include <stdio.h>
#include <conio.h>
```

```
void main(void)
{
int i,n;
int x[20],y[20],z[20];

clrscr();
printf("n=");
scanf("%d",&n);

i=0;
do
{
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
}while(i++<n);
i=0;
do
{
        printf("y[%d]=",i);
        scanf("%d",&y[i]);
}while(i++<n);
i=0;
do
{
        z[i]=x[i]+y[i];
}while(i++<n);
i=0;
do
{
        printf("\nz[%d]=%d",i,z[i]);
}while(i++<n);
getch();
}
/*****************************************************************
*****************************************************************
***************************/
/*
program care realizeaza scaderea a doi vectori de aceeasi
dimensiune, versiune cu for

*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
```

```
int i,n;
int x[20],y[20],z[20];

clrscr();
printf("n=");
scanf("%d",&n);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }

for(i=0;i<n;i++)
        {
        printf("y[%d]=",i);
        scanf("%d",&y[i]);
        }
for(i=0;i<n;i++)
        {
        z[i]=x[i]-y[i];
        }
for(i=0;i<n;i++)
        {
        printf("\nz[%d]=%d",i,z[i]);
        }
getch();
}
/*************************************************************************
*************************************************************************
***************************/
/*
program care realizeaza produsul scalar a doi vectori de aceeasi
dimensiune, versiune cu for

*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
int i,n,s=0;
int x[20],y[20];

clrscr();
printf("n=");
```

```
scanf("%d",&n);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }

for(i=0;i<n;i++)
        {
        printf("y[%d]=",i);
        scanf("%d",&y[i]);
        }
for(i=0;i<n;i++)
        {
        s+=x[i]*y[i];
        }
printf("\nProd scalar este %d",s);
getch();
}

/*************************************************************************
*************************************************************************
***************************/
/*
program care realizeaza ordonarea descrecatoare
prin "Buble sort" a unui vector de aceeasi dimensiune, versiune cu for

*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
int i,j,n,t;
int x[20];

clrscr();
printf("n=");
scanf("%d",&n);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }
```

```
for(i=0;i<n-1;i++)
        {
        for(j=i+1;j<n;j++)
                {
                if(x[i]<x[j])
                        {
                        t=x[i];
                        x[i]=x[j];
                        x[j]=t;
                        }
                }
        }
printf("Vectorul ordonat descrescator este: \n");
for(i=0;i<n-1;i++)
        {
        printf(" %d",x[i]);
        }
getch();
}
/*************************************************************************
*************************************************************************
***************************/
/*
program care realizeaza ordonarea crescatoare
prin "Buble sort" a unui vector de aceeasi dimensiune, versiune cu for

*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
int i,j,n,t;
int x[20];

clrscr();
printf("n=");
scanf("%d",&n);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }
for(i=0;i<n-1;i++)
        {
```

```
        for(j=i+1;j<n;j++)
                {
                if(x[i]>x[j])
                        {
                        t=x[i];
                        x[i]=x[j];
                        x[j]=t;
                        }
                }
        }
printf("Vectorul ordonat crescator este: \n");
for(i=0;i<n-1;i++)
        {
        printf(" %d",x[i]);
        }
getch();
}
/*********************************************************************
*********************************************************************
**************************/
/*
program care realizeaza sumarea a doi vectori de
dimensiuni diferite, evident se considera completat cu zero vectorul de
dimensiune mai mica

*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
int i,n,m,dim;
int x[20],y[20],z[20];

clrscr();
printf("n=");
scanf("%d",&n);
printf("m=");
scanf("%d",&m);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }
```

```
for(i=0;i<m;i++)
        {
        printf("y[%d]=",i);
        scanf("%d",&y[i]);
        }
if(n<m)
{
for(i=0;i<m;i++)
        {
        if(i<n)z[i]=x[i]+y[i];
        else z[i]=y[i];
        }
dim=m;
}
else
{
for(i=0;i<n;i++)
        {
        if(i<m)z[i]=x[i]+y[i];
        else z[i]=x[i];
        }
dim=n;
}

for(i=0;i<dim;i++)
        {
        printf("\nz[%d]=%d",i,z[i]);
        }
getch();
}
/*************************************************************************
*************************************************************************
***************************/
/*
program care realizeaza scaderea a doi vectori de
dimensiuni diferite, evident se considera completat cu zero vectorul de
dimensiune mai mica

*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
int i,n,m,dim;
int x[20],y[20],z[20];
```

```
clrscr();
printf("n=");
scanf("%d",&n);
printf("m=");
scanf("%d",&m);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }

for(i=0;i<m;i++)
        {
        printf("y[%d]=",i);
        scanf("%d",&y[i]);
        }
if(n<m)
{
for(i=0;i<m;i++)
        {
        if(i<n)z[i]=x[i]-y[i];
        else z[i]=-y[i];
        }
dim=m;
}
else
{
for(i=0;i<n;i++)
        {
        if(i<m)z[i]=x[i]-y[i];
        else z[i]=x[i];
        }
dim=n;
}

for(i=0;i<dim;i++)
        {
        printf("\nz[%d]=%d",i,z[i]);
        }
getch();
}
/*************************************************************************
*************************************************************************
***************************/
```

```
/*
program care realizeaza cautarea unei valori intr-un vector,
si anunta prima pozitie pe care este gasita valoarea
*/
#include <stdio.h>
#include <conio.h>

void main(void)
{
int i,n,t,val,pos=-1;
int x[20];

clrscr();
printf("n=");
scanf("%d",&n);

for(i=0;i<n;i++)
        {
        printf("x[%d]=",i);
        scanf("%d",&x[i]);
        }
printf("Valoarea de cautat:");
scanf("%d",&val);
i=0;
while((x[i]!=val)&&(i++<n));
if(i<n)pos=i;
if(pos>0)printf("El se gaseste in pozitia a %d a",pos+1);
else puts("Nu s-a gasit in vector");
getch();
}
```

Seturi de teme

SET1

1. Scrieți un program care afişează un mesaj.
2. Scrieți o funcție care întoarce argumentul la puterea a treia cu prototipul
   int pw3 (int num).
   X= pw3 (y) <-> $x = y^3$.
3. Scrieți o funcție ce calculează $x = y^4$ folosind funcția sqrt(double x)
4. Scrieți un program care citeşte de la tastatură un întreg şi-l afişează la puterea a treia prin folosirea pw3.
5. Scrieți în pseudocod un program care elimină comentariile dintr-o sursă C.
6. Eliminați erorile din următoarea secvență:

```
main (x)
    {
    scanf ( ” Câți gândaci sunt ?“ , nr_gândaci );
    printf ( “ Acest program are  %d probleme “, nr_gândaci )ş
    }
```

SET2

1. Atunci când se afişează o variabilă de tip float sau double folosind specificatorul de format %f câte cifre zecimale sunt afişate ?
   printf() realizează rotunjirea sau trunchierea valorii afişate.
2. Scrieți un program ce conține următoarele declarații:

```
char c;
int j;
float f;
```

   şi afişează adresa fiecărei variabile. Vorbiți despre modul în care compilatorul alocă memorie pentru variabile.
3. Scrieți echivalentul octal, hexa, zecimal pentru următoarele numere binare:
   1. 0001.0010
   2. 0110.0101
   3. 0110.1011
   4. 1011.1011
   5. 0011.1111
   6. 0101.1111
   7. 0000.0010.0110.0100

   în ambele cazuri când se consideră că numerele sunt date în
   a) complement față de 1
   b) complement față de 2
4. Scrieți declarațiile pentru următoarele formate:
   a) Întreg lung fără semn
   b) Variabilă în virgulă mobilă cu dublă precizie
   c) Pointer la caracter
   d) Caracter inițializat cu ‘x’
   e) Funcție externă ce întoarce un întreg fără semn

5. Scrieți următoarele numere în binar în complement față de 1 sau în complement față de 2:
   1, -1, 255, 256, 511, 512, 513, 128, -128, 0xFF, 0x7F.

SET3

1. Fiecare calculator este limitat în precizia maximă care o poate folosi în reprezentarea numerelor în virgulă mobilă. Din acest motiv pentru un epsilon suficient de mic următoarea expresie va fi adevărată.

$$1.0 = 1.0 + eps$$

Să se scrie un program care găseşte cea mai mare valoare pentru eps pe computerul pe care e executat.

Obs.: această valoare diferă între float şi double. Se va folosi 1.0 pentru teste şi nu 0.0 pentru că majoritatea calculatoarelor nu au instrumente de lucru specifice pentru manipularea aritmeticii lui 0.

2. Să se rescrie următorul program fără a se folosi instrucțiunile break, continue şi go to.

```
/* programul numără caracterul 'a' din intrare
#include <stdio.h>
#include <ctype.h>
void main (void)
{
        int nr_a=0;
        char c;
        c=getch();
        while(1)
        {
                if(c== '\n') break
                if (isdigit(c)) continue
                if (c=='a') goto add_a;
                get_ch: c=getchar();
                goto end;
                add_a: nr_a++;
                goto get_ch;
                end: ;
        }
}
```

3. Scrieți o funcție care acceptă un întreg la intrare şi scrie un număr de spații egal cu pas de intrare. Folosind aceeaşi funcție să se scrie un program care citeşte caractere de la intrarea standard, şi le afişează dar înlocuieşte taburile cu 5 spații.
4. Multe programe care necesită transfer de caractere dintr-un loc în altul folosesc un mecanism de checksum (sumă de control) pt. a verifica corectitudinea transferului. Aceasta se realizează prin sumarea valorilor tuturor caracterelor transmise şi trimiterea ei împreună cu şirul respectiv. Dacă suma se potriveşte, există o oarecare probabilitate ca ele să fi fost transferate corect. Scrieți un program care să folosească o funcție

unsigned int CS (char *s)

care să calculeze checksum-ul pentru o linie introdusă de la tastatură.

5. Scrieți un program care clasifică caracterele citite într-una din următoarele categorii:
   white _Space (space, '\n', '\r', '\t')
   punctuator (",!;:()?)
   alfabet (a-z , A-Z)
   numeric (0-9)
   uncknown (oricare alt caracter)
   Deci la orice tastă atinsă, să apară pe ecran codul tastei şi clasa din care face parte.
   5.1. Programul să fie rezolvat numai cu if, else şi return
   5.2. Programul să fie rezolvat numai cu switch şi return.
6. Scrieți un program care afişează literele mici şi mari precum şi valorile lor zecimale.

SET 4

1. Introduceți paranteze în expresiile de mai jos a.î.să fie specificat (cu ajutorul lor) explicit modul de evaluare implicit folosit de compilator.(Se va folosi tabela de precedență)
   a) a=b*c==2
   b) a=f(x)&&a>100
   c) a==b&&x!=y
   d) a=b+=2+f(2)
   e) a=b>>2+4
   f) a=s.f+y.x
   g) a=b&&a>z?x=y:z
   h) a=*++p*p
   i) a=b^c&d

2. Să se construiasca o expresie ce va genera un long int (4 octeți) prin apelul repetat al funcției getbt() care întoarce un singur octet. Ordinea de aranjare este ca primul apel al lui getbt() va genera octetul cel mai semnificativ din întreg (nr=Expresie (getbt()).

3. Care este ieşirea următorului program ?

```
   void main(void)
{
    short i=0;
   printf ("y.d\n",(i+1)*(1=1));
 }
```

4. Să se scrie o funcție unsigned long int circular_shift (unsigned long int a,int n); care face deplasare stanga cu recirculare , cu n poziții a numărului a.

   a=0001.0110.0011.1010.0111.0010.1110.0101

aplicând circular_shift (a,5) spre stânga =>
1100.0111.0100.1110.0101.1100.1010.0010

5. Folosind <<sau>> să se determine care este cel mai mare întreg care poate fi reprezentat pe calculator.
6. Scrieți o funcție care primeşte un întreg şi-l afişează în binar (folosiți sizeof (int) pentru a afla nr de biți.
7. Scrieți o funcție care citeşte un nr în forma binară şi-l converteşte în formă hexa.
8. Scrieți o funcție pack care compune un long int din 4 caractere de intrare long int pack (char a,char b, char c, char d)

Bublle sort

```
# define FALSE 0
#define TRUE 1
#include <stdio.h>
void bubble_sort (int*v,int n)
{
    int j,k,temp,sorted=FALSE;
    while (!sorted)
    {
        sorted=TRUE  //sc pp vect sortat
        for (j=0;j<n-1;j++)
        {
            if (v[j]>v[j+1])
            {
                // cel puțin un element nu este în ordine
                sorted =FALSE;
                temp=v[j];
                v[j]=v[j+1];
                v[j+1]=temp;
            }
        }
    }
}

int strlen1(char *s)
{
    int i=0;
    while (s[i])++i;
    return i;
}//întoarce nr elemente dintr-un nr mai puțin '\0'
```

```
void strcopy1(char *s1, char *s2)
{
    int i;
    for (i=0;*(s1+1];++I)*(s2+i)=*(s1+I);
    s2[++I]='0';
}  //classical
```

sau

```
void strcopy1(char *s, char *s2)
{
    while (*s2++=*s1++);
}    //professional
```

echivalentul funcției pas din Pascal , str str un contor un nr într-altul ….

```
Int strstr1(char *s1,char *s2)
{
    char *p,*q, *subs;
    for (subs=s1;*subs; subs++)
    {
            p=subs;
            q=s2;
            while (*q)
            if(!(*q++!=*p++)) return subs –s1;
            else break;
    }
    return( –1);
} //de verificat
```

Set 5

1. Să se explice ce va rezulta în urma evaluării următoarelor expresii:

   Fie
   ```
   static int ar []={10,15,4,25,3,-4}
   int *p;
   p=&ar [2];
   ```

   a) *(p+1)
   b) p[-1]
   c) (ar-p)
   d) ar [*p++]
   e) *(ar+ar[2])

2. Ce este greşit în următoarul caz:

   ```
   Int j , v[5]={1,2,3,4,5}
   ……
   for (j=1; j<5; ++j)
   printf("x.d\n",v[j])
   ```

3. Modificați bubble sort a.î. în loc să rearanjeze elementele într-un tablou el să stocheze numai ordinea corectă de aranjare.

Deci v[]={$\overset{0}{13}, \overset{1}{25}, \overset{2}{11}, \overset{3}{2}, \overset{4}{14}$} în urma aplicării bubble sort modificarea se realizează în tabloul

ord[] = {1,4,0,2,3}

Se observă :

V[ord[0]]=V[1]=25
V[ord[1]]=V[4]=14
V[ord[2]]=V[0]=13
.
.

4. Scrieți o funcție
   void sort_concat(double *a, double *b, double *c)
   care ia a,b două tablouri sortate şi le depune în c care trebuie să fie şi el sortat deasemenea.
5. Modificați funcția de mai sus astfel încât să elimine valorile egale.
6. Scrieți o funcție
   char * plussir (char *a, char *b);

care concatenează pe a cu b.

7. Declarațiile
   char S[10]
   şi
   char *S
   sunt la fel dacă nu arătați diferențele scriind un program în care ele nu pot fi înlocuite.
8. Scieți o funcție care sortează un tablou de şiruri de caractere în ordine alfabetică.
   ex. : S[][] = {{mike},{oana},{lili},{dana}}
   în urma apelului funcției S[][] = {{dana},{lili},{oana},{mike}}.
9. Fiind date următoarele declarații :
   static int a[2][3]={{-3,14,5},{1,-10,8}};
   static int *b[]={a[0],a[1]};
   int *p=b[1];
   Cum vor fi evaluate urm[toarele expresii :
   a) *b[1]
   b) *(++p)
   c) *9*(--p-2)
10. Care din următoarele expresii este echvalentă cu a[j] ? :
    1. *(a[j]+k)
    2. **(a[j+k])
    3. (*a+j))[k]
    4. (*(a+k))[j]
    5. *((*a+j))+k)
    6. **(a+j)+k
    7. *(&a[0][0]+j+k)

Set 6

1. Scrieţi o funcţie care afişează pe ecran din 20 în 20 de apeluri , de câte ori a fost apelata în total.
2. Să se scrie 2 funcţii :

➢ conversie integer -> BCD
➢ conversie BCD -> integer

folosindu-se câmpurile de biţi (bit field).

3. O stivă este o listă mai particulară în sensul că :
   ➢ adăugarea se poate face doar la capătul listei (operaţie numită push)
   ➢ ştergerea mai tot din coada listei (operaţie numită pop)

   Scrieţi un program care :
   ➢ crează o listă de acest tip folosind cele 2 funcţii
     push pentru adăugare
     pop pentru ştergere.

   Trebuiesc implementate şi aceste funcţii.

Set7

1. Să se scrie un program care să primească în linia de comandă o frază precum şi switch-urile –c sau –C funcţie de acestea, programul vă oferă fraza din linia de intrare cu litere mici sau mari. Dacă switch nu este precizat se va considera implicit –c adică afişarea cu litere mici.
2. Scrieţi o versiune recursivă a funcţiei strlen. Este recursivitatea mai bună ca versiunea normală. Argumentaţi.
3. Scrieţi o funcţie recursivă care calculează cel mai mare divizor comun a două nr. pozitive întregi.
4. Scrieţi şi o versiune iterativă pentru punctul 3.
5. Scrieţi o funcţie recursivă care acceptă un pointer la string ca argument, transformă respectivul şir într-o listă înlăanţuită de caractere şi întoarce pointerul către primul element din listă.
6. Scrieţi o funcţie recursivă care numără elementele dintr-o listă înlănţuită. Argumentul trebuie să fie pointerul către primul element din listă. Evident că valoarea returnată va fi întreagă.
7. Scrieţi o funcţie recursivă care afişează valorile fiecărui element dintr-o listă înlănţuită.
8. Verificaţi legalitatea următoarelor declaraţii:
   a. *(*( )) [ ]
   b. * (**) [ ]
   c. (*(*(*x( )) [ ]) ( ))
   d. **x[ ] ( )
   e. * (x[ ]) [ ]
   f. * (* (x ( )) ( ))

Set8

1. Explicaţi efectele (şi traducerea) următoarelor macrouri:
   1.1. #define BS 1024
        int buf [BS+1]

1.2. #define a[b] b+1
…
a(1) + 1

1.3. # define a(b) b+1
…
a(1) +1

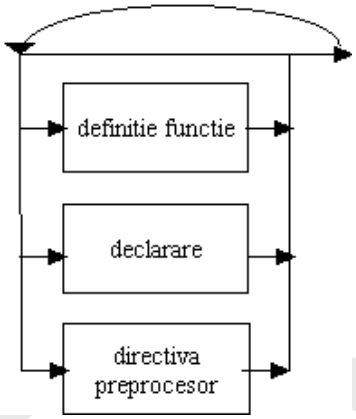
1.4. #define cos(x) *cos(x)
…
cos(x) + cos (cos(y) + 1)

1.5. #define min(x,y) ((x) >= (y) ? x:y)
min (1,min (a,b))

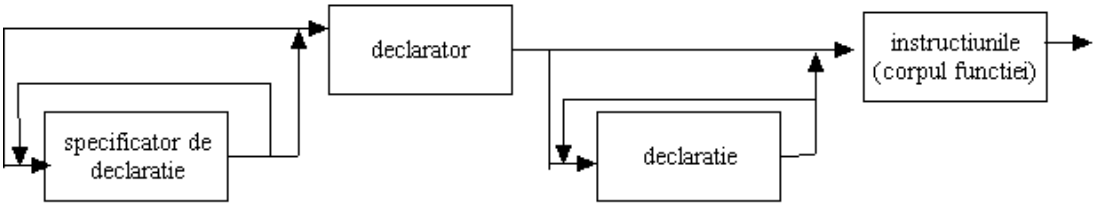
Set9

1. Scrieți un program care să implementeze comanada
   # include “nume_fişier”
2. Scrieți o funcție care elimină toate spațiile suplimentare dintr-o sursă C
3. Scrieți un program care să verifice inchiderea corectă a parantezelor într-o sursă C. Indicație: stiva.
4. Scrieți un program care numără caracterele, cuvintele şi liniile unui fişier.
5. Scrieți un program care primeşte ca argumente mai multe fişiere şi le afişează pe ecran (un fel de type cu mai multe inturi)

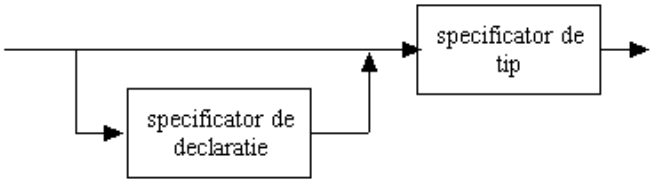
file:



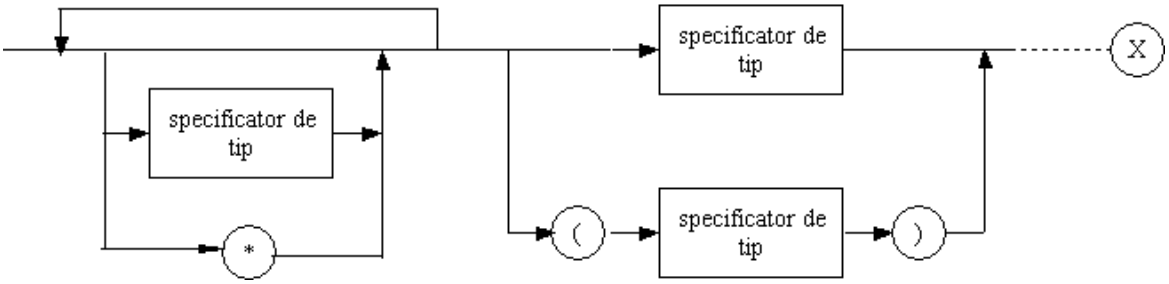
Definirea unei functii:

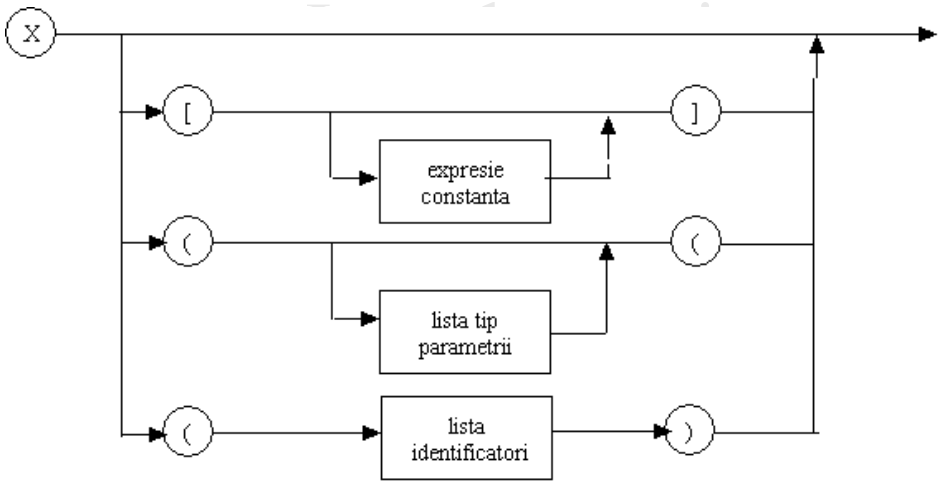


Specificator de declaratie

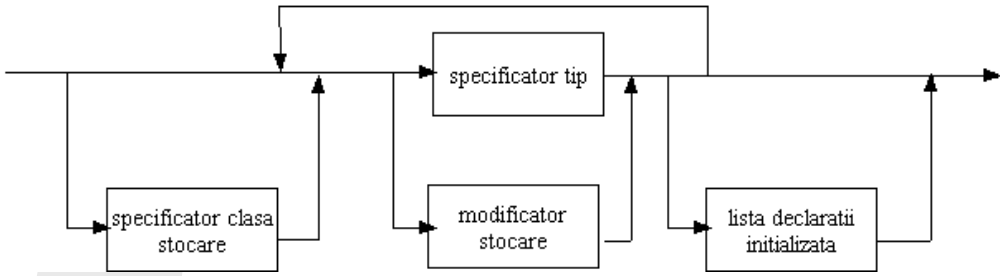


Declarator:

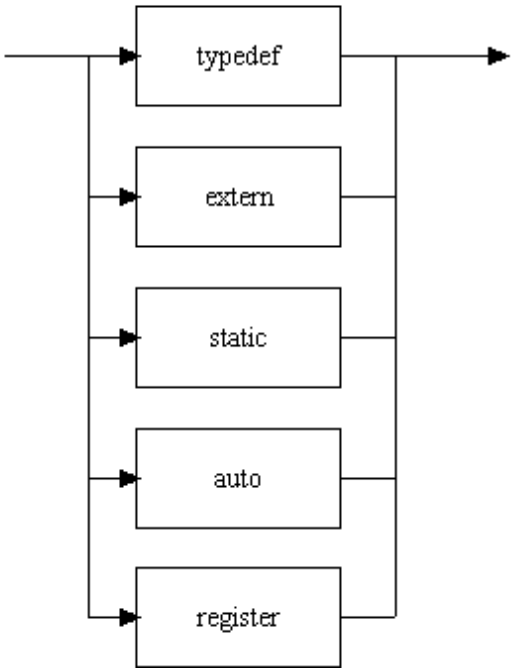




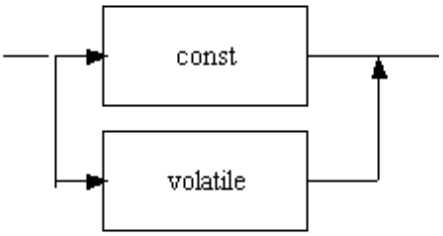
Declaratii:



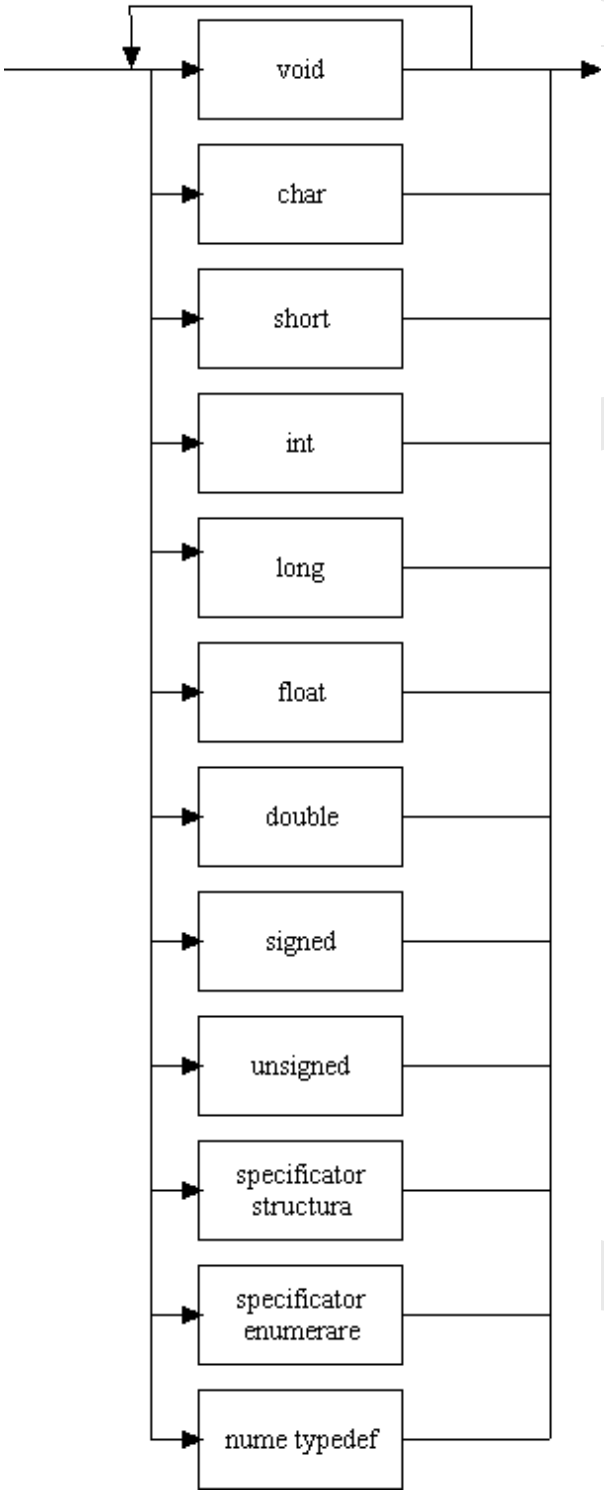
Specificator clasa de stocare:



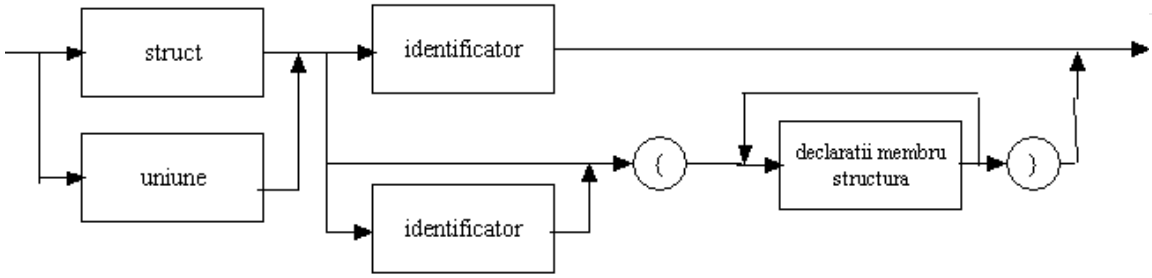
Modificator stocare:



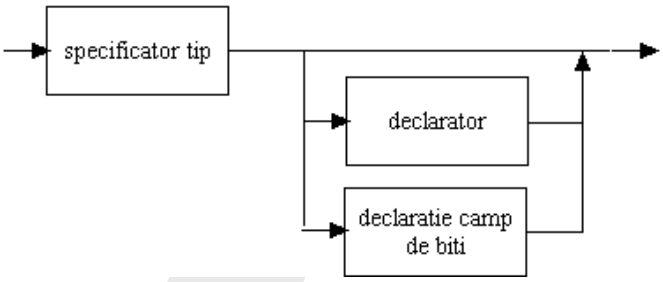
Specificator tip:



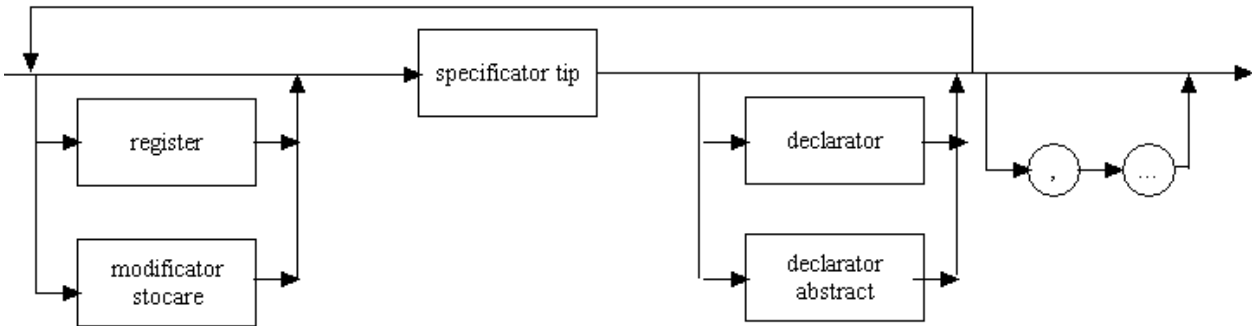
specificator de structura sau uniune



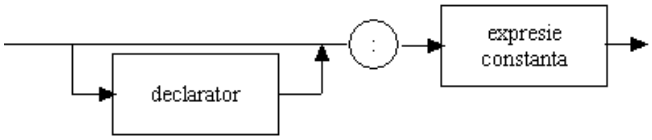
declaratii membru structura



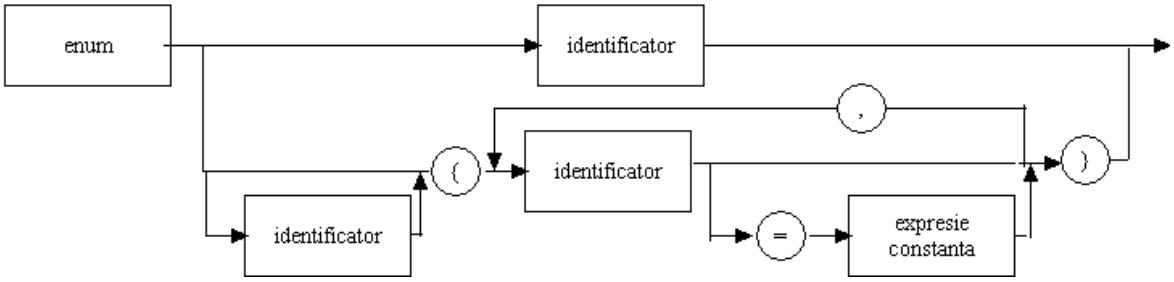
lista cu tipul parametrilor:



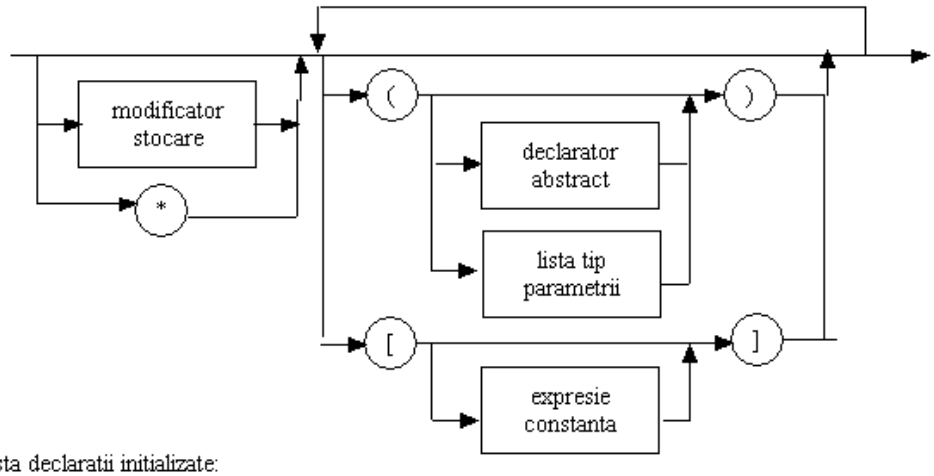
declaratii camp de biti



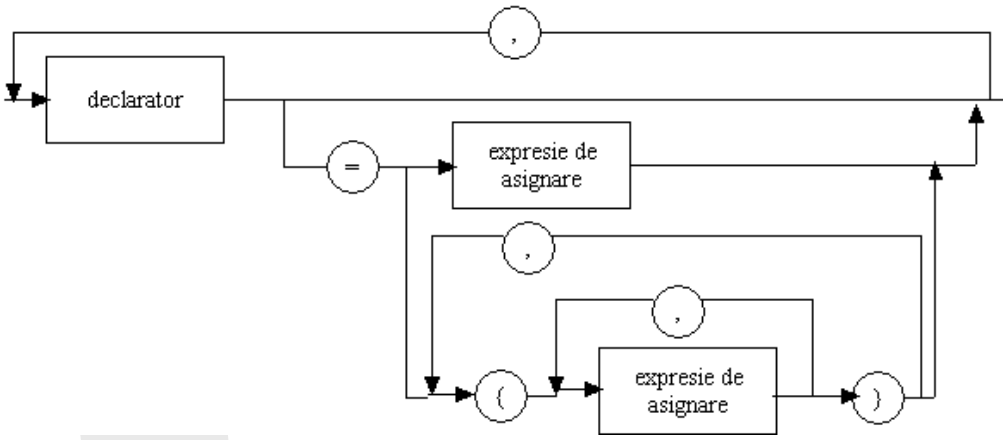
specificator de enumerare:



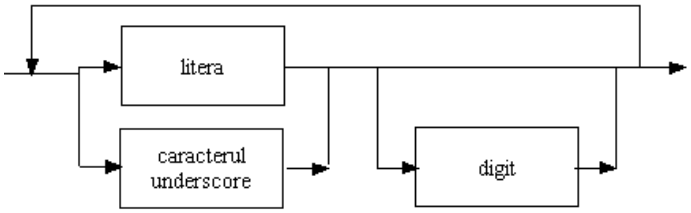
declarator abstract



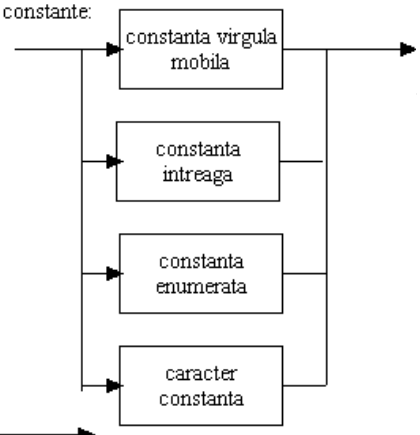
lista declaratii initializate:



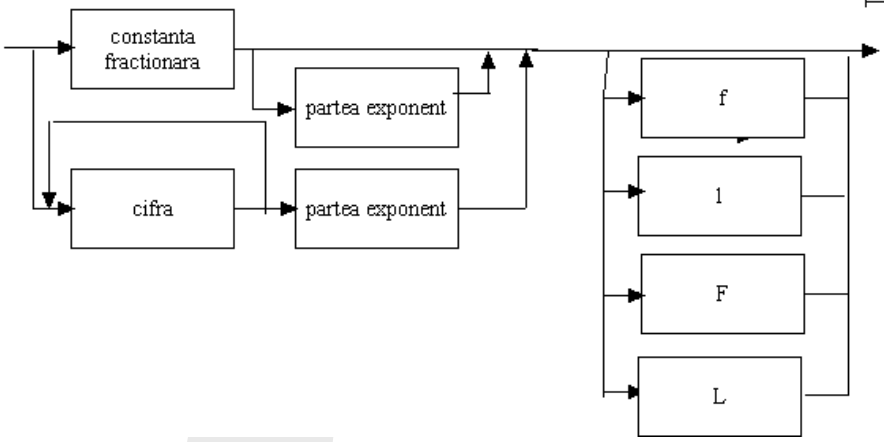
identificator:



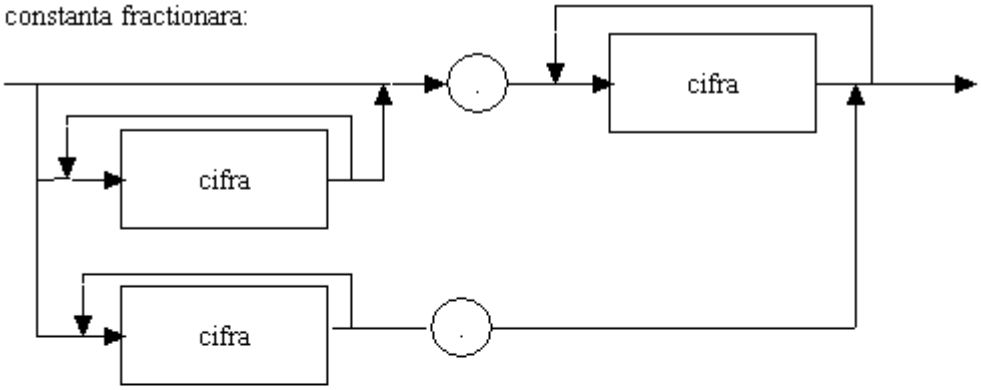
constante:



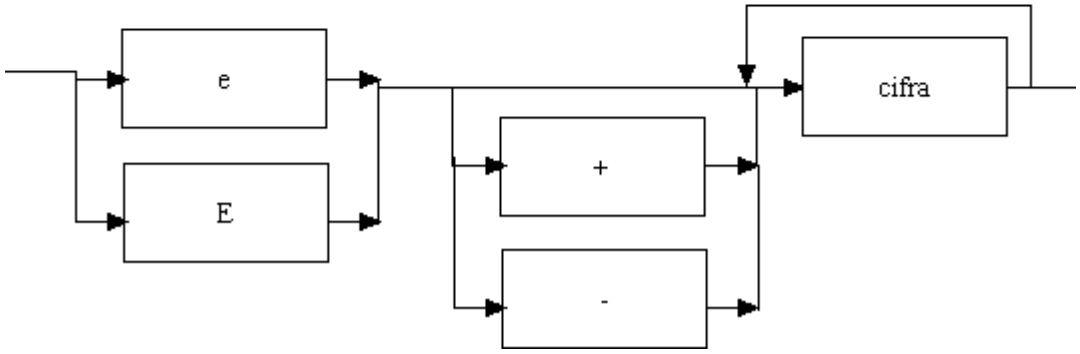
constanta in virgula mobila:



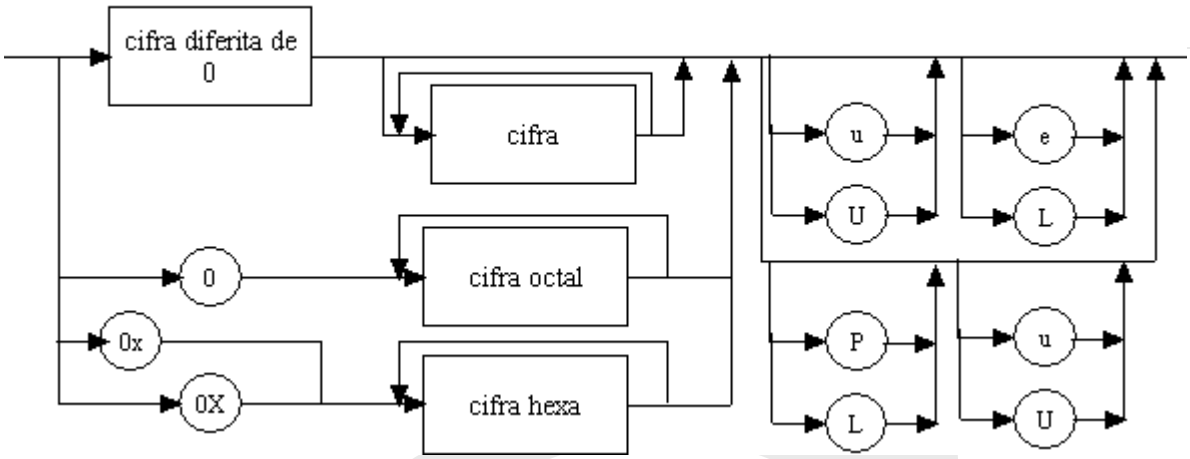
constanta fractionara:



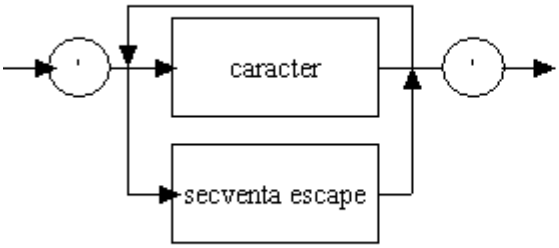
partea exponent:



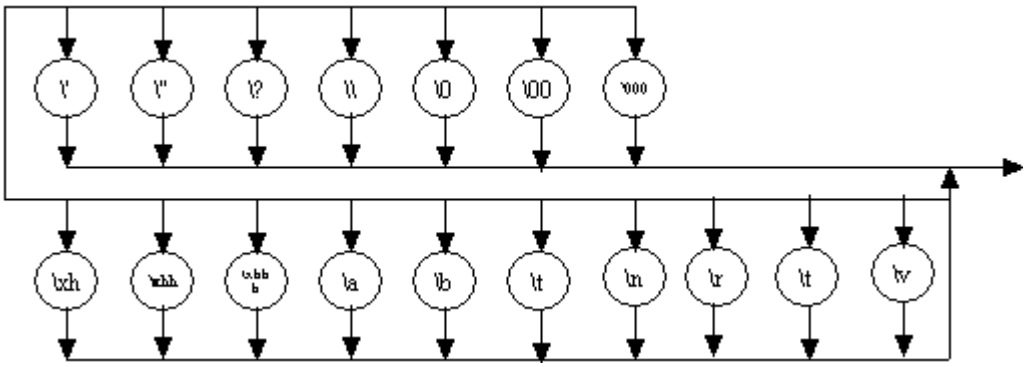
constanta intreaga:



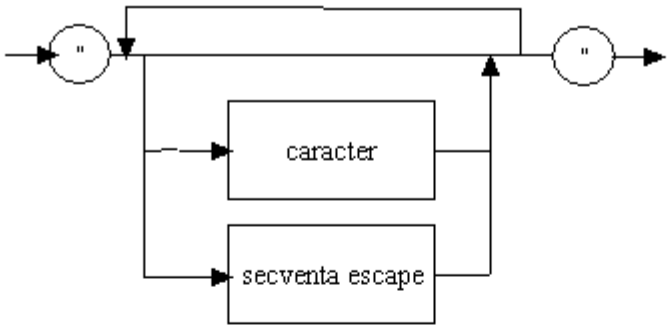
constanta caracter:



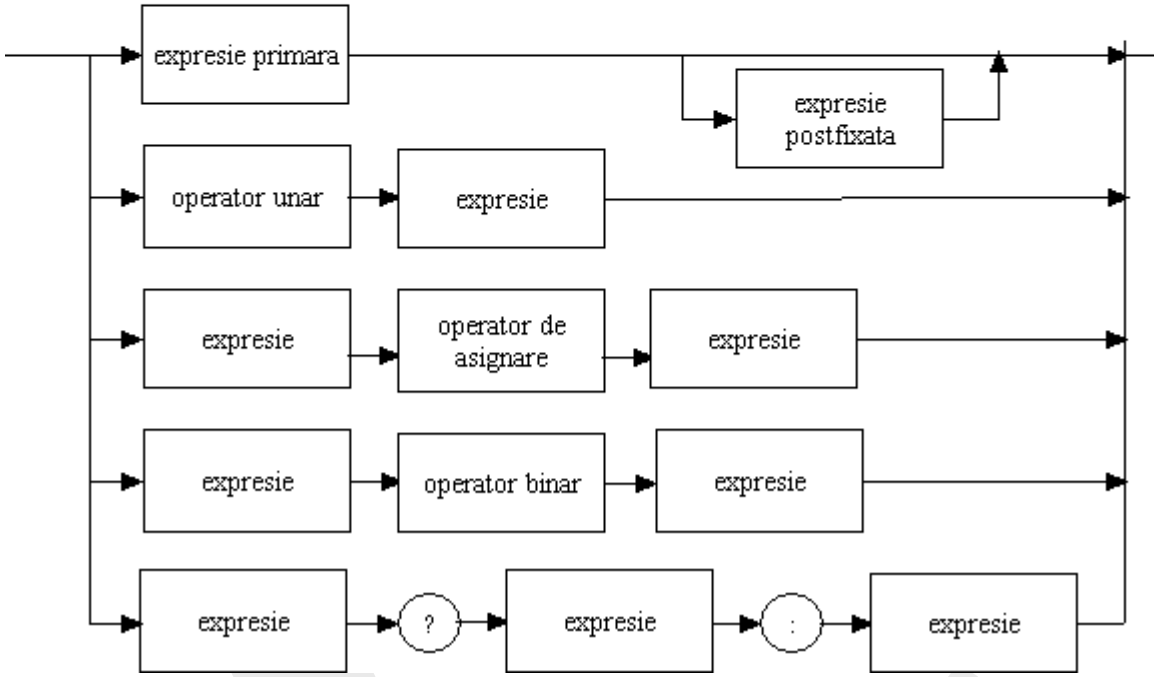
secventa escape



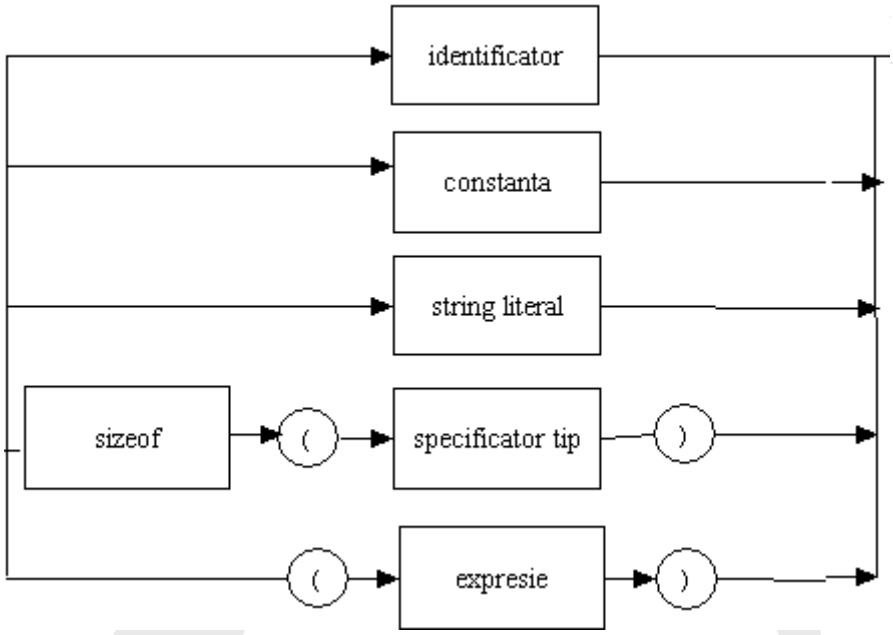
string literal:



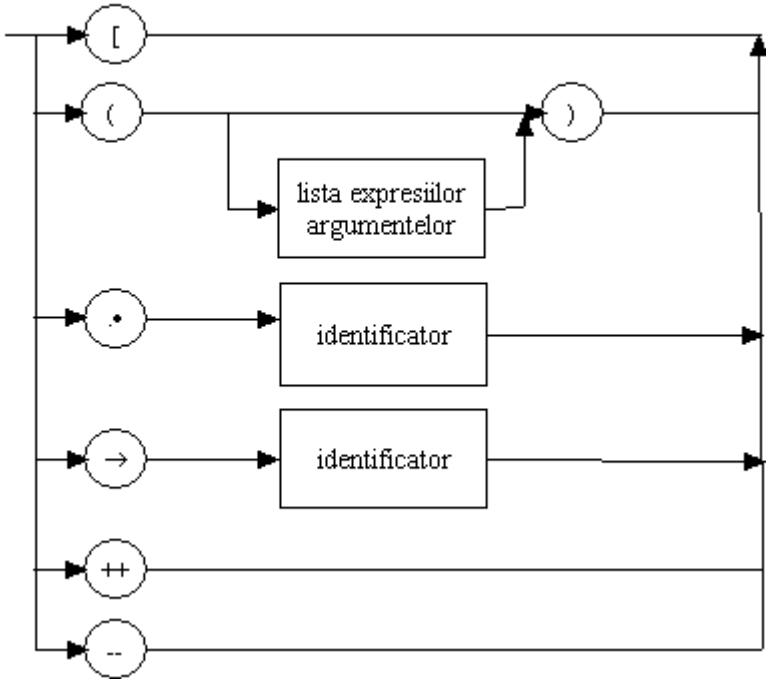
expresie:



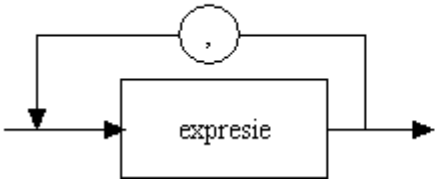
expresie primara:



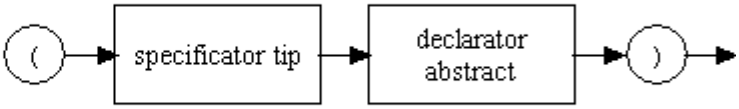
expresie postfixata:



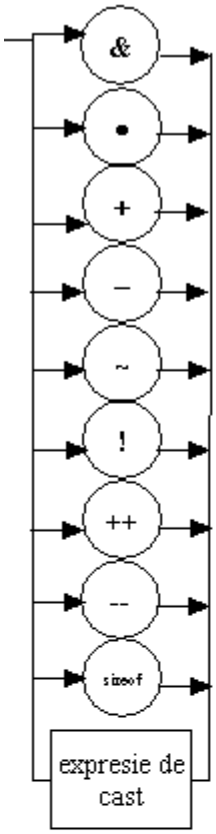
lista expresiilor argumentelor:



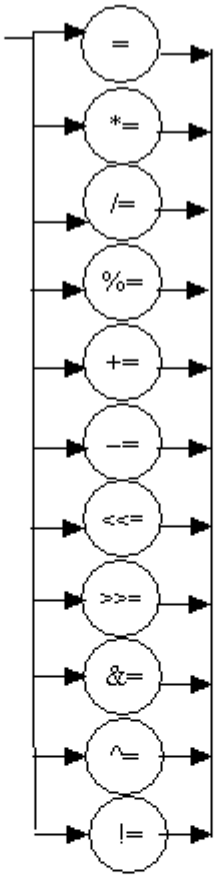
expresie de cast:



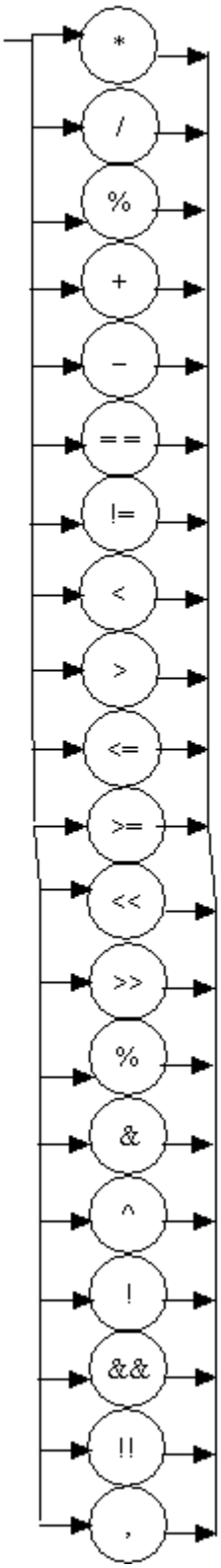
operator unar:



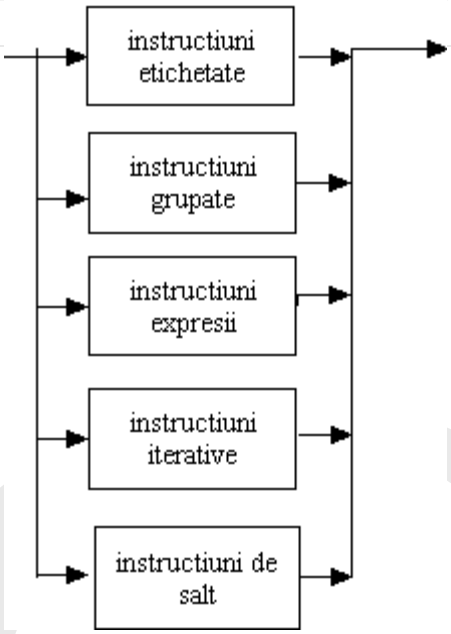
opearatori de asignare:



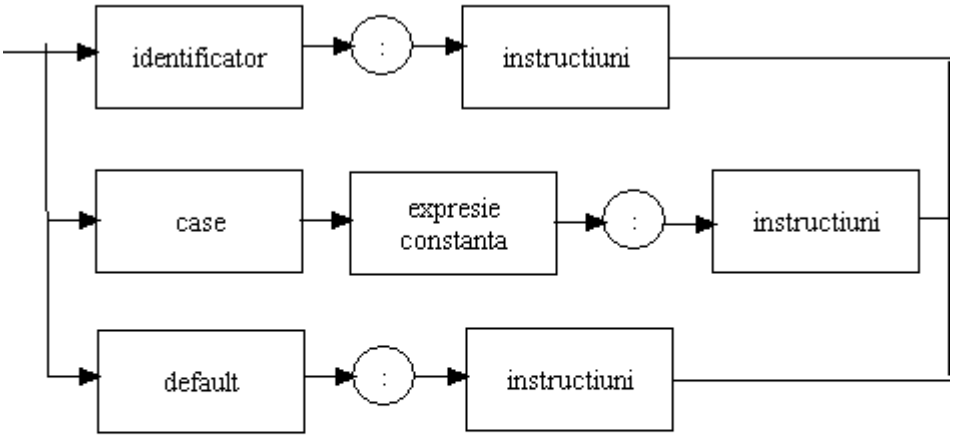
Operatori binari:



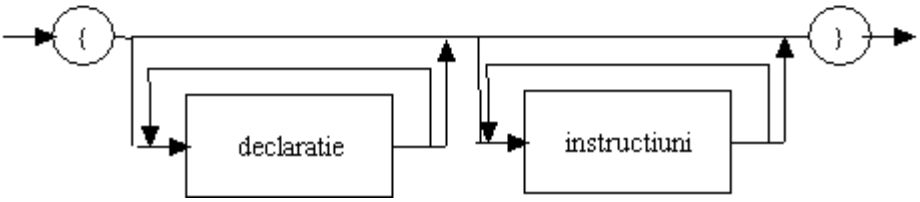
instructiuni:



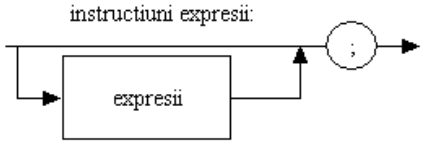
instructiuni etichetate:



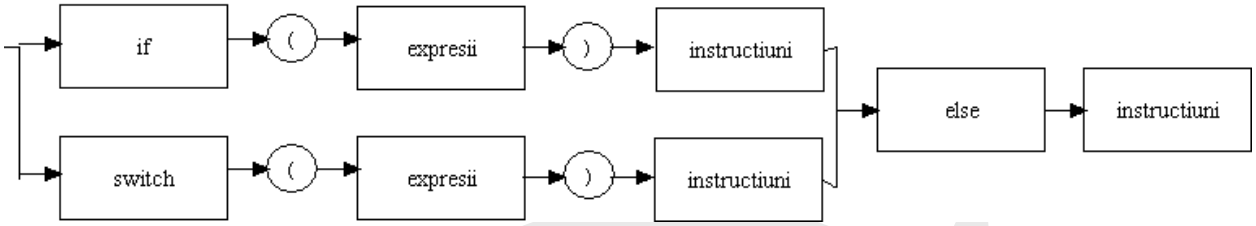
instructiuni grupate:



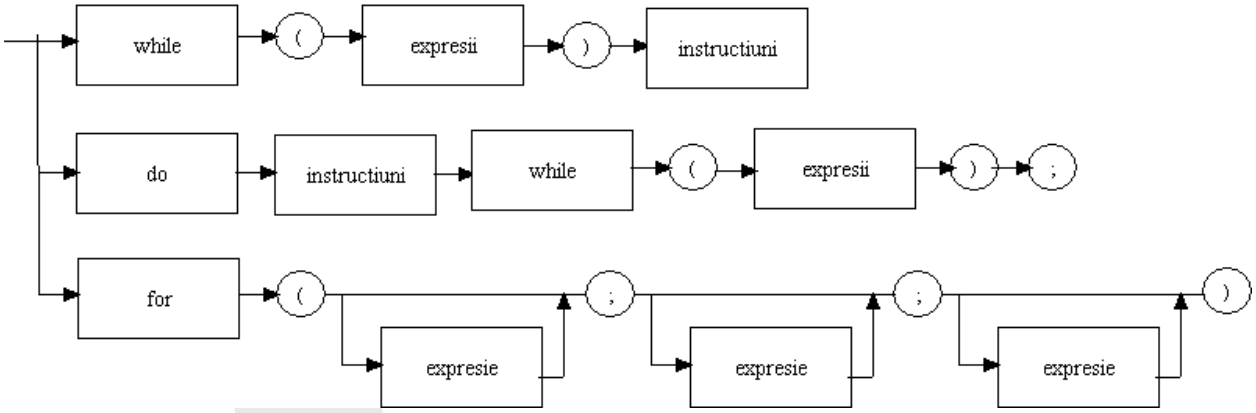
instructiuni expresii:



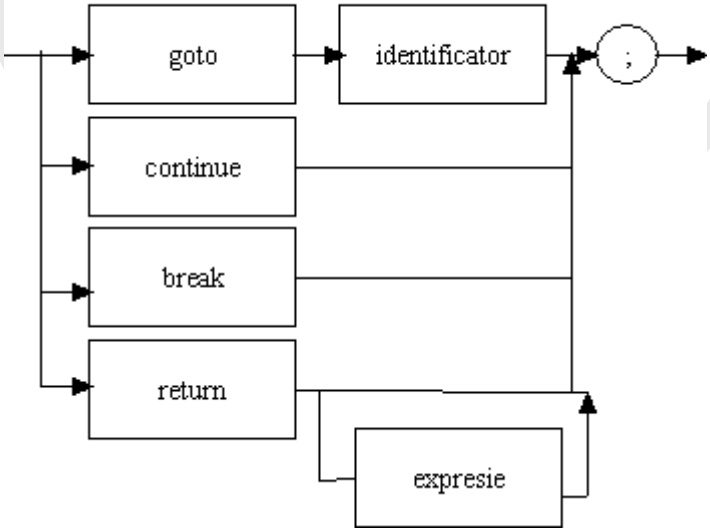
instructiuni de selectie:



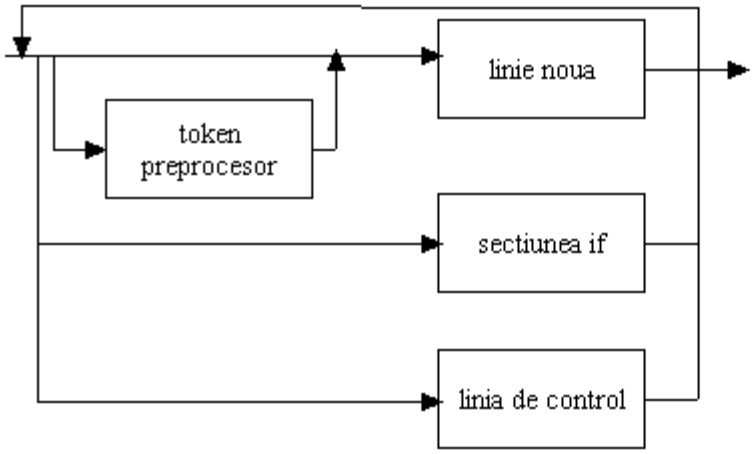
instructiuni iterative:



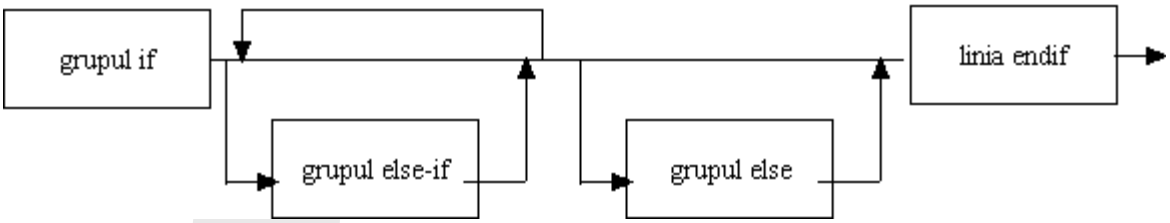
instructiuni de salt:



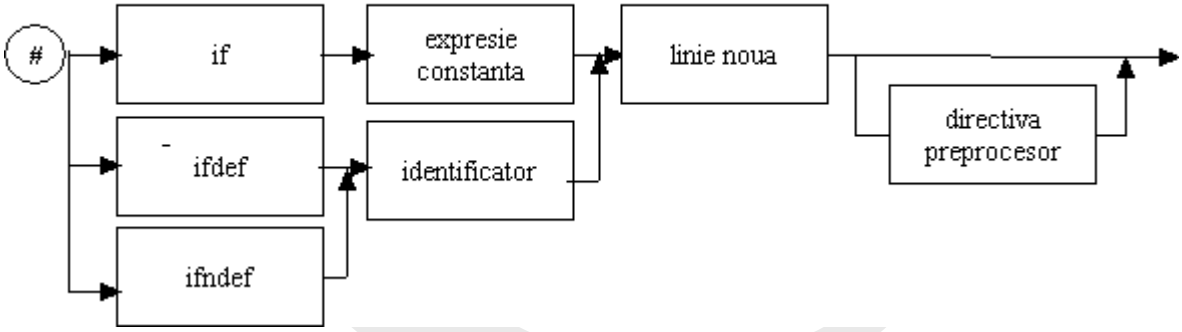
directiva preprocesor:



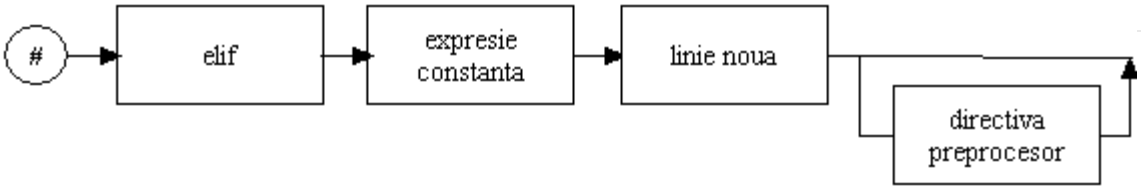
sectiunea if:



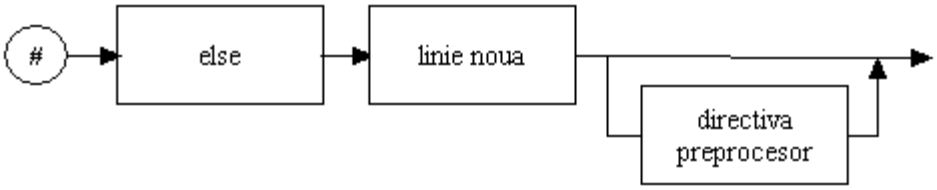
grupul if



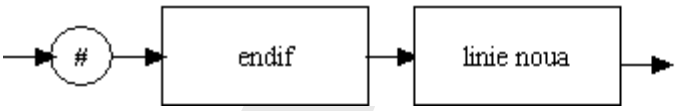
grupul elif:

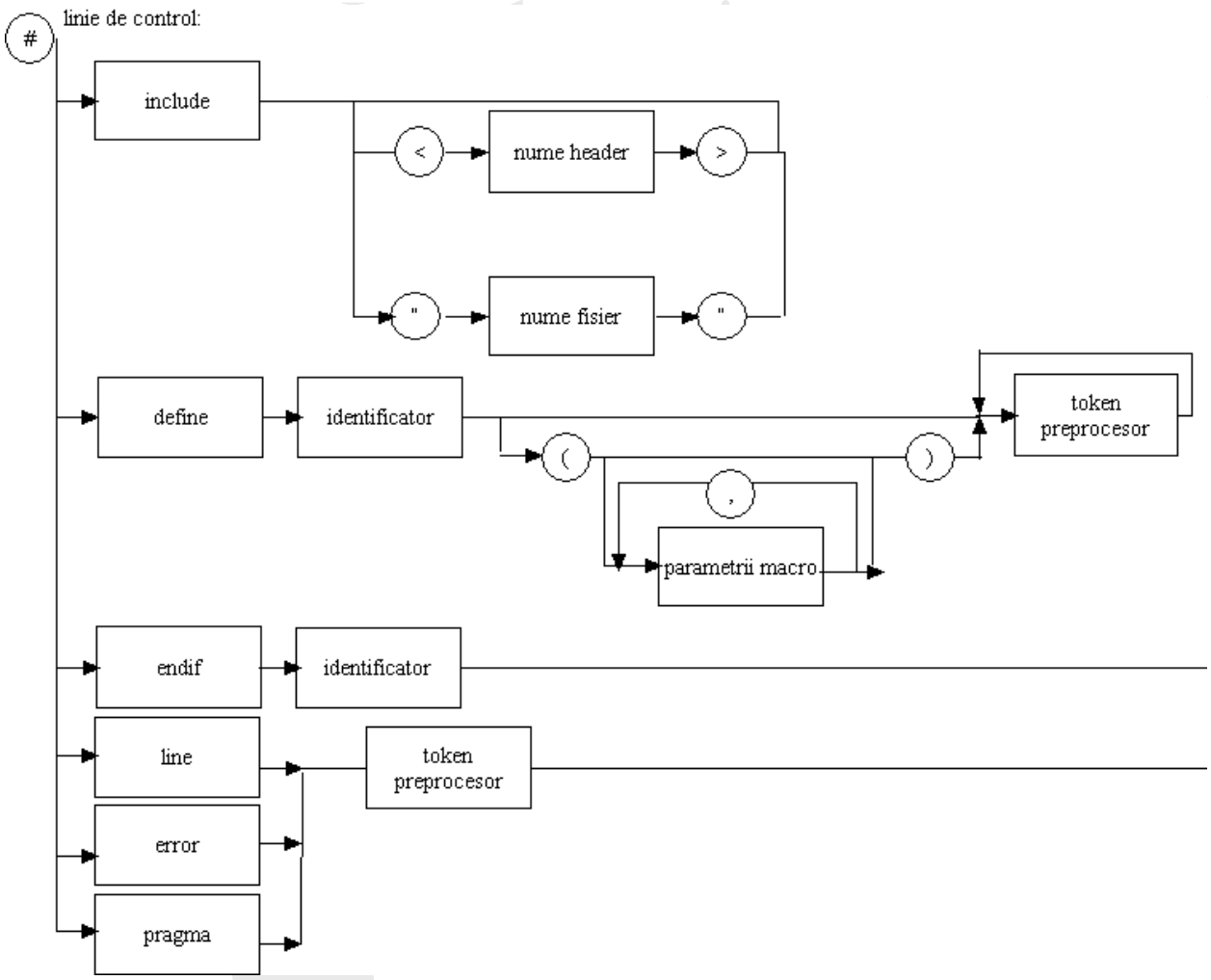


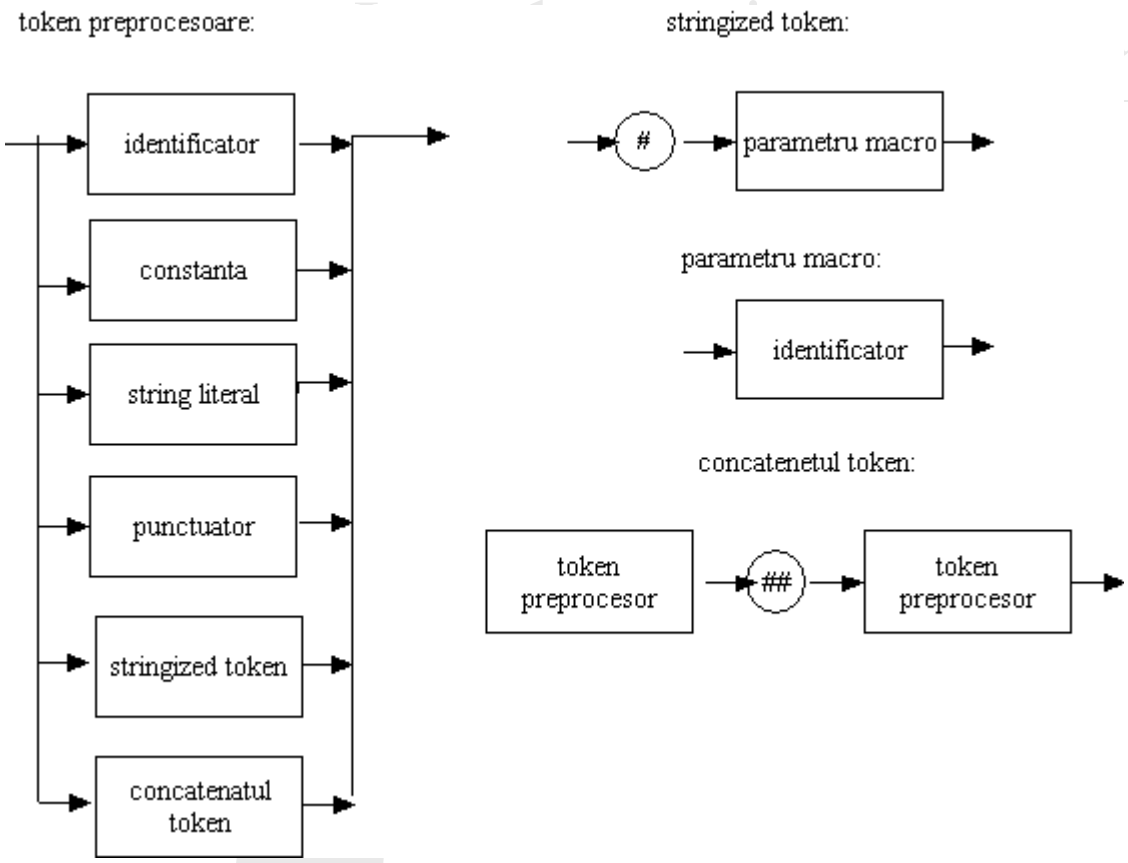
grupul else:



linie endif:




#
linie de control:
include
<
nume header
>
"
nume fisier
"
define
identificator
(
,
parametrii macro
)
token
preprocesor
endif
identificator
line
error
pragma
token
preprocesor


token preprocesoare:
identificator
constanta
string literal
punctuator
stringized token
concatenatul token
stringized token:
#
parametru macro
parametru macro:
identificator
concatenetul token:
token preprocesor
##
token preprocesor

Cuprins

Capitolul 1
1 Introducere.....1
1.1 Programarea si limbaje de programare.....3
1.2 Arhitectura unui sistem de calcul……..……3
1.3 Functionarea unui sistem de calcul ……..……4
1.4 Notiunea de algoritm ……...……..6
1.5 Reperezentarea algorimilor……..……7
1.5.1 Metoda shemei logice……..……...7
1.5.2 Metoda pseudocodului……...8
Capitolul 2
2 Limbajul C……9
2.1 Structura generica a unui program scris in C……10
2.2 Etapele generarii unui program prornid de la o problema reala……10
Capitolul 3
3 Notiunea de variabila si tipuri de variabile……....15
3.1 Initializarea variabilelor……..16
3.2 Tipuri de variabile……...17
Capitolul 4
4 Tipuri de date predefinite……...18
4.1 Date numerice……18
4.1.1 Operatorul virgula ……...19
4.2 Date alfanumerice……...19
4.2.1 Caractere speciale de control…….....20
4.3 Operatiuni simple de citire sau afisare a datelor……20
4.3.1 Functii de intrare sau iesire cu utilizatorul……...21
Capitlul 5
5 Operatori ai limbajului C……...24
5.1 Operatori aritmetici……24
5.2 Expresiile ……...24
5.2.1 Ordinea de evaluare a expresiei……...25
5.3 Tipuri aritmetice de date ……26
5.4 Operatorii unari ++ si -- ……27
5.5 Operatorii op=……27
5.6 Operatorii relationari si logici……27
5.7 Expresii logice complexe……28
Capitolul 6
6 Instructiuni de selectie(decizie)……31
6.1 Instructiune simpla de decizie……31
6.2 instructiune de decizie multipla……..35
Capitolul 7
7 Instructiuni de ciclare. Introducere……38
7.1 Instructiuni cu un numar necunoscut de pasi……..38

Introducere in programare

7.1.1 Instructiunea while……...38
7.1.2 Instructiunea do .. while ……..39
7.2 Instructiuni cu numar cunoscut de pasi ……..41
7.3 Alte instructiuni……..43
7.3.1 Instructiunea break……..43
7.3.2 Instructiunea continue……43
7.3.3 Instructiunea goto nume_eticheta……43
7.3.4 Instructiunea return……..44
7.3.5 Constante simbolice……44
Capitolul 8
8 Masive de date. Tablouri……45
8.1 Tablouri monodimensionale…..……45
8.1.1 Prelucrari simple cu vectori……45
8.1.2 Siruri de caractere……48
8.2 Tablouri bidimensionale. Matrici……49
Capitolul 9
9 Tipuri de date definite de utilizator……51
9.1 Declararea unei structuri……51
9.2 Operatorul typedef……..51
9.3 Accesarea membrilor unei structuri……52
9.4 O structura in interiorul altei structuri……53
9.5 Structuri autoreferite……...54
9.6 Transferul structurilor ca argumente de functii……..54
9.7 Returnarea unei structuri……54
9.8 Campuri de biti……...55
9.9 Uniuni……55
9.10 Enumerari……59
Capitolul 10
10 Notiunea de functie……61
10.1 Declararea si definirea functiilor……62
10.1.1 Declararea functiilor……62
10.1.2 Declaratii implicite……63
10.2 Definirea functiilor……...63
10.3 Tipul de data void……66
10.4 Recursivitatea……....73
10.5 Variabile globale……...77
10.6 Macrouri……78
10.7 Macrouri predefinite……79
10.7.1 ASSERT……79
10.7.2 Macrouri specializate in clasificarea caracterelor……80
10.7.3 Macrodefinitii predefinite……81
Capitolul 11
11 Pointeri si alocare dinamica……83
11.1 Introducere……...83
11.1.1 Notiunea de pointer……...83
11.1.2 Operatii aritmetice cu pointeri……...85
11.1.3 Modele de memorie……...86

Introducere in programare

11.2 Alocare dinamica……..87
11.2.1 Introducere……..87
11.2.2 Alocarea dinamica……88
11.2.3 Alocarea dinamica a tablourilor de date……90
11.2.3.1 Alocarea tablourilor monodimensionale……91
11.2.3.2 Alocarea tablourilor bidimensionale……...92
11.2.3.3 Alocarea tablourilor tridimensionale……..93
11.3 Pointeri la structuri……94
11.3.1 Accesul membrilor……94
11.3.2 Alocarea structurilor de date……..94
11.3.3 Initializare si distrugere structuri……...94
11.4 Declaratii complexe in C……..96
11.5 Descifrarea declaratiilor complexe in C……...96
11.6 Lucrul cu fisierele de date (I/O FILE) Streamuri……100
11.6.1 Citirea si scrierea datelor……101
11.6.2 Lucrul la nivel de character……..101

Capitolul 12

12 Facilitati de nivel scazut ale limbajului C……..104
12.1 Functii pentru accesul direct la memorie……104
12.2 Functii pentru accesul la porturile de intrare sau iesire……..105
12.3 Functii pentru gestionarea intervalelor de timp……..105
12.4 Functii pentru controlul sistemului de intreruperi……..105
12.5 Functii pentru alocarea memoriei(prin DOS)……106
12.6 Intreruperi software. Accesul la registrele procesorului……106
12.7 Functii pentru citirea sau modificare vectorilor de intrerupere……..107
12.8 Apeluri de functii DOS……...108
12.9 Functii pentru terminarea sau lansarea programelor……...110
12.10 Dezvoltarea de cod in limbaj mixt(C si ASM)……112
12.11 Tratarea dispozitivelor bus asincrone……...113

Capitlul 13

13 Preprocesorul……115
13.1 Directiva nula……115
13.2.Directivele #define si #undef……..……115
13.3 Directiva #include……...116
13.4 Directiva de compilare conditionata……...116
13.5 Directiva line……..116
13.6 Directiva #error……...117
13.7 Directiva #pragma……..117

Exemple……121
Seturi de teme……..143
Sintaxa ANSI……151